Computer Science at Kent

YATL: Yet Another Transformation Language - Reference Manual Version 1.0

Octavian Patrascoiu

Technical Report No. 2-04
March 2004

This report presents version 1.0 of YATL (Yet Another Transformation Language), which is evolving in order to support all the features provided by [QVT02] and the future QVT standard. The first subsection provides a quick overview of the YATL language. Subsequent sections present the features of YATL in more details.

Chapter 1. YATL OVERVIEW

YATL is a hybrid language (a mix of declarative and imperative constructions) designed to answer the Query/Views/Transformations Request For Proposals [QVT02] issued by OMG and to express model transformations as required by the MDA [MDA] approach.

YATL formulates queries to interrogate the model using constructions from the OCL 2.0 standard. A YATL query is a syntactic construct that wraps inside the description of the request in terms of OCL 2.0 (see **Appendix 3**). The YATL processor invokes the OCL processor to process the query and supply the results of interrogation.

A YATL transformation describes a mapping between a source MOF metamodel S, and a target MOF metamodel T. The transformation engine uses the mapping to generate a target model instance conforming to T from a source model instance conforming to S. The source and the target metamodels may be the same metamodel. Navigation over models is specified using OCL.

Each transformation contains one or more transformation rules. A transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS of a YATL transformation is specified using a filtering expression written either in OCL or native code such as Java, C#, and scripts. This approach allows filter expressions to include both modeling information (e.g. navigational expressions, properties values, collections) and platform dependent properties (e.g. special conversion functions), which makes them extremely powerful. A compound statement specifies the effect of the RHS. The LHS and RHS for the YATL transformation are described in the same syntactical construction, called transformation rule. A rule is invoked explicitly using its name and with parameters.

The abstract syntax of YATL namespaces, translation units, queries, views, transformations, and transformations rules is described in Figure1.

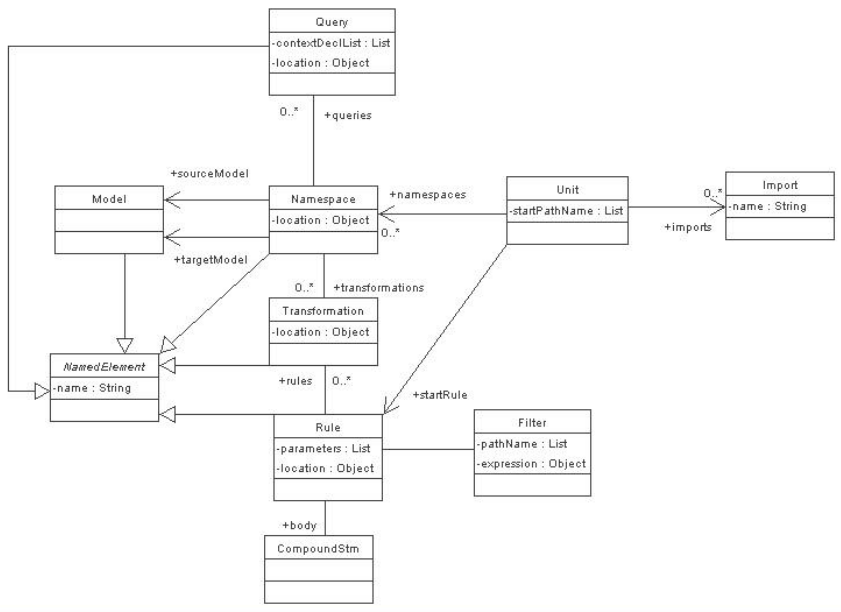

Figure1 Abstract Syntax

Chapter 2. MAIN FEATURES

The declarative features come mainly from OCL expressions and the description of the LHS of transformation rules. YATL acts in a similar way to a database system that uses SQL to interrogate the database and the imperative host language to process the results of the query. We choose OCL to describe the matching part of YATL rules because it is a well defined language for querying the UML models it provides a standard library with an acceptable computational expressiveness, it is a declarative language, and it is a part of the OMG's standards.

YATL supports several kinds of imperative features, used in the RHS of transformation rules, which are presented later in this chapter. This features were selected so that YATL can provide lifecycle operations like creation and deletion, operations to change the value of properties, declarations, decisions, and iteration statements, native statements to interact to the host machine, and build statements to ease the construction of target model instance. Compound statements contain a sequence of instructions, which are to be executed in the given order. These syntactic constructions make use of OCL expressions to specify basic operations such as adding two integer values. YATL uses the same type system as OCL 2.0 [OCL].

YATL is described by an abstract syntax (a MOF metamodel) and a textual concrete syntax. It does not yet have a graphical concrete syntax as QVT RFP suggested. A transformation model in YATL is expressed as a set of transformation rules. Transformations from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) can be written in YATL to implement the MDA.

A YATL transformation is unidirectional. We believe that a model transformation language should be unidirectional, otherwise it cannot be used for large scale models. The main difficulty with a bidirectional transformation language is that it needs some reasoning to perform the transformation. For example, DSTC's proposal [QVTD] uses mechanisms similar to Prolog-unification to perform a bidirectional mapping. The reverse transformation can be described as any other transformation using YATL.

For a real model-to-model transformation, traceability is absolutely necessary to make the approach workable. To trace the mapping between source and target model instances, YATL comprises an operator called *track*. Track expressions are, from the concrete syntax point of view, similar to DSTC's track constructions [QVTD]. The main difference is that YATL's tracks are defined using concepts like relation name, domain, and imagine, and not Prolog-like concepts (e.g. unification). This approach makes the traceability system of YATL suitable for large-scale systems.

Chapter 3. PROGRAMS

A YATL *program* consists of one or more source files, known formally as *translation units*. A source file is an ordered sequence of Unicode standard characters. Conforming implementations must accept Unicode source files encoded with the UTF-8 encoding form [UNI], and transform them into a sequence of Unicode characters. Implementations may choose to accept and transform additional character encoding schemes, such as UTF-16, UTF-32, or non-Unicode character mappings.

Conceptually speaking, a YATL program is analysed in five steps:

(1) Character conversion, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.

(2) Lexical analysis, which translates a stream of Unicode input characters into a sequence of tokens.

(3) Syntactic analysis, which translates the sequence of tokens into an abstract representation of the input structure.

(4) Semantic analysis, which checks if the input follows the semantic rules, and produces an internal representation of both syntax and semantics.

(5) Code generation or interpretation where the semantic representation is either used to generate code for the underlying machine or directly evaluated on the same machine.

Chapter 4. GRAMMARS

This section presents the syntax of YATL language using two grammars, structured on two levels. On the first level, the *lexical grammar* defines how Unicode characters are combined to form line terminators, white space, comments, and YATL tokens. At the second level, the *syntactic grammar* defines how the tokens resulting from the lexical grammar are combined to form YATL programs. Both grammars are described using the notation comprised in **Appendix 1**.

4.1. Lexical grammar

The lexical grammar of YATL is presented in **Appendix 2**. The terminal symbols of the lexical grammar are the characters of the Unicode character set, and the lexical grammar specifies how characters are combined to form white spaces, comments, and tokens.

The lexical processing of a YATL source file consists of reducing the file into a sequence of tokens that becomes the input to the syntactic analysis. Line terminators, white space, and comments can serve to separate tokens, but otherwise these lexical elements have no impact on the syntactic structure of a YATL program.

When several lexical grammar productions match a sequence of characters in a source file, the lexical processing always forms the longest possible lexical element. For example, the character sequence ‑ is processed as the beginning of a single-line comment because that lexical element is longer than a single – token.

Every source file in a YATL program must conform to the *input* production of the lexical grammar.

4.2. Syntax grammar

The syntactic grammar of YATL is presented in **Appendix 3** and the following sections. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined to form YATL programs.

Every source file in a YATL program must conform to the *translation-unit* production of the syntactic grammar.

Chapter 5. TYPES AND VARIABLES

The types of the YATL language are derived from the OCL's types [OCL2], [AP03], [ALP03]. They can be used to encapsulate logical values, numbers, collections, tuples, and user types. The type hierarchy of YATL is described in Figure1 and derives from [ALP03].

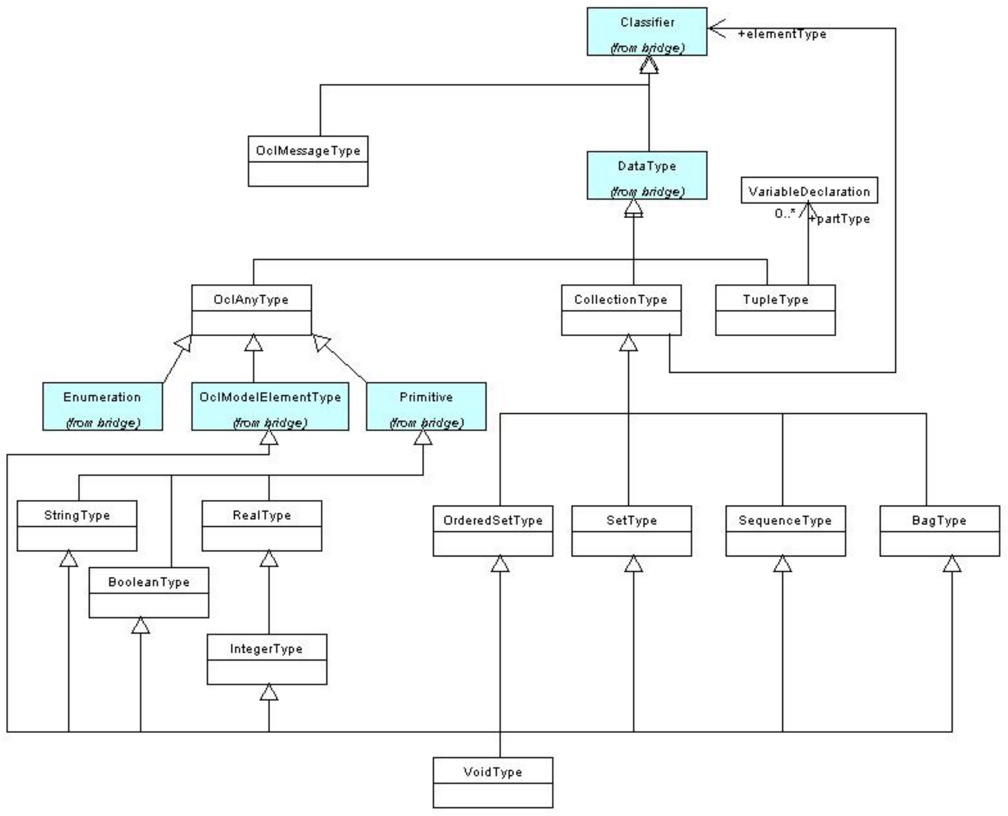

Figure1 YATL types

YATL's type system is unified such that a value of any type can be treated as a *Classifier.* Every type in YATL directly or indirectly derives from the *Classifier* class type, which is the ultimate base class of all types. On the other hand, undefined values are represented using *VoidType*.

YATL defines two categories of variables: local variables and value parameters. In the example

```
transformation T {
      rule r match java::Class (String s) {
            let i: Integer = 3;
      }
}
```

s is a value parameter and *i* is a local variable.

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. YATL is a type-safe language, and the YATL processor guarantees that values stored in variables are always of the appropriate type. The value of a variable can be changed through assignment. If the value of a variable is not specified by an initialization or assignment, it is considered to be the undefined value from OCL.

A variable must be *definitely assigned* before its value can be obtained. A variable is said to be *definitely assigned* at a given location in the executable code, if the compiler can prove, by a particular static flow analysis that the variable has been automatically initialized or has been the target of at least one assignment.

Variables are either *initially assigned* or *initially unassigned*. An initially assigned variable has a well defined initial value and is always considered definitely assigned. An initially unassigned variable has no initial value. For an initially unassigned variable to be considered definitely assigned at a certain location, an assignment to the variable must occur in every possible execution path leading to that location.

5. EXPRESSIONS

This section defines the syntax, order of evaluation of operands and operators, and meaning of expressions. YATL expressions are extensions of OCL 2.0 expressions presented in Figure2 [ALP03].

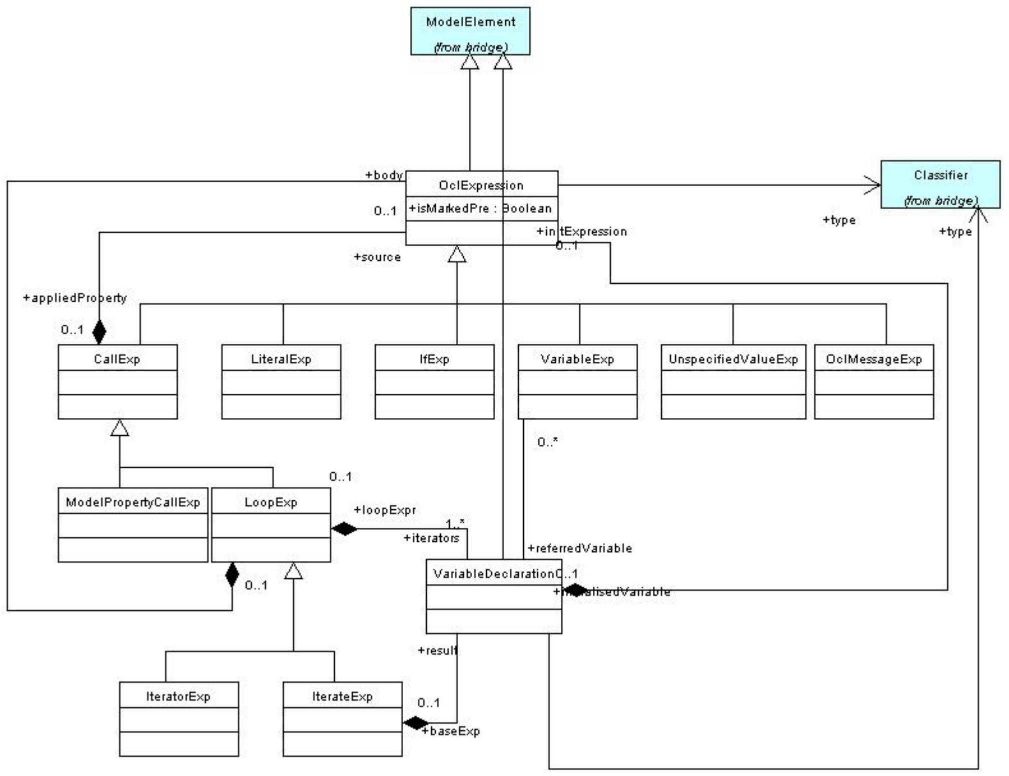

Figure2 YATL expressions

More details about the expressions supported by OCL (e.g. concrete syntax, abstract syntax, and semantics) and the way they are implemented can be found in [OCL2][ALP03].

The extensions specific to YATL are presented in the following subsections.

5.1. The assignment operator

The assignment operator assigns a new value to a variable or a property.

assignment-expression →

ocl-expression ':=' rhs-expression .

rhs-expression →

ocl-expression |

new-expression |

build-expression |

track-expression .

The left operand of an assignment must be an expression classified as a variable or a property.

In an assignment, the right operand must be an expression of a type that is compatible to the type of the left operand [OCL2]. The operation assigns the value of the right operand to the variable or property given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

5.1.1. The *new* operator

The *new* operator is used to create new instances of model element types [OCL2].

new-expression →

'new' path-name .

The *new* operator implies creation of an instance of the *path-name* type.

5.1.2. The *build* operator

The *build* operator is used to create new instances of model element types and set their properties in the same time.

build-expression →

'build' path-name '{' list-pair '}'.

list-pair→

λ |

pair ‘,’ list-pair .

pair →

name ‘:=’ rhs-expression .

The *new* operator implies creation of an instance of the *path-name* type and sets the values for the properties specified in *list-pair*. If there is at least one *name* for which there is no such property in type *path-name,* a compile-error is reported.

5.1.3. The *track* operator

The track operator is used to store and retrieve mappings during and after the transformation process.

track-expression →

‘track’ ‘(‘ ocl-expression ‘,’ simple-name ‘,’ ocl-expression ‘)’ |

‘track’ ‘(‘ ‘null’ ‘,’ simple-name ‘,’ ocl-expression ‘)’ |

‘track’ ‘(‘ ocl-expression ‘,’ simple-name ‘,’ ‘null’ ‘)’ .

Given a relation R and two objects X and Y, the meaning of the track operator is the following:

- *track(X, R, Y)* stores the relation R(X, Y).
- *Y := track(X, R, null)* retrieves the element related to X.
- *X := track(null, R, Y)* retrieves the element related to Y.

The type of X and Y can be any OCL 2.0 type (e.g. integer, real, boolean, string, model element type, collection, or tuple).

Chapter 6. STATEMENTS

This section contains the description of the statements supported by YATL and other basic concepts such as: end point, reachability, name lookup, rule resolution etc. The abstract syntax tree of YATL statements is described in Figure 6.1.

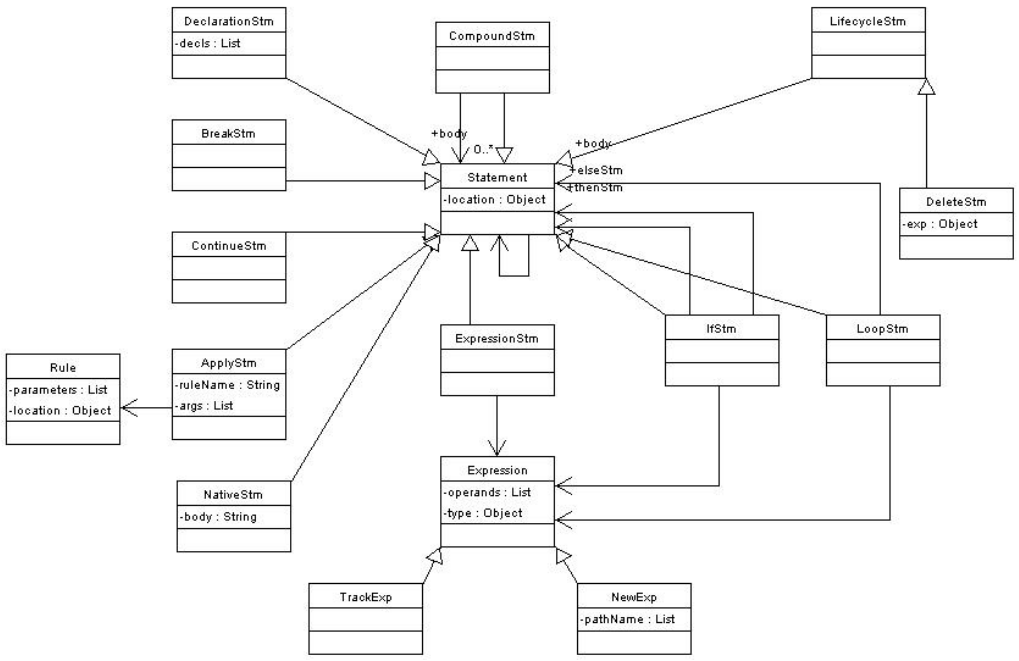

Figure 6.1 YATL statements

6.1.1. End points and reachability

Every statement has an *end point*. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be *reachable*. Conversely, if there is no possibility that a statement will be executed, the statement is said to be *unreachable*. In the example

```
rule r() {

     while ( … ) {

      -- reachable
      let i: Integer = 3;
      break;

      -- unreachable
       i := i+1;

   }

}
```

6.1.2. Blocks

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

block →

'{' '}'

|

'{' statement-list '}' .

A *block* consists of an optional *statement-list*, enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements. The scope of a local variable or constant declared in a block is the block. Within a block, the meaning of a name used in an expression context must always be the same.

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

6.1.3. Statement lists

A *statement-list* consists of one or more statements written in sequence. Statement lists occur in *block*s.

> *statement-list* →
>
> *statement* |
>
> *statement-list   statement* .

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.

The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

6.2. The empty statement

An *empty-statement* does nothing.

> *empty-statement*→
>
> *';'.*

An empty statement is used when there are no operations to perform in a context where a statement is required.

Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

6.3. Declaration statements

A *declaration-statement* declares a local variable. Declaration statements are permitted in blocks.

declaration-statement→

local-variable-declaration .

6.3.1. Local variable declarations

A *local-variable-declaration* declares one or more local variables [OCL2], [ALP03].

local-variable-declaration →

'let' variable-declaration-list ';'

variable-declaration-list →

variable-declaration |

variable-declaration-list ',' variable-declaration .

variable-declaration →

simple-name [':' type] ['=' init-expression] .

The *type* of a *local-variable-declaration* specifies the type of the variables introduced by the declaration [OCL2][ALP03]. The *init-expression* gives the initial value of the variable. Both type and initial value are optional [OCL2].

The value of a local variable is obtained in an expression using a *simple-name*, and the value of a local variable is modified using an *assignment*. A local variable must be definitely assigned at each location where its value is obtained.

The scope of a local variable declared in a *local-variable-declaration* is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local-variable-declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

The example

```
rule r() {
     let x : Integer = 1,
         y : Integer,
         z : Integer = x * 2;
}
```

corresponds exactly to

```
rule r() {
     let x : Integer;
      x := 1;
     let y : Integer;
     let z : Integer;
      z := x * 2;
}
```

6.4. Expression statements

An *expression-statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

expression-statement →

expression ';' .

expression →

assignment-expression |

ocl-expression |

track-expression .

Execution of an expression statement evaluates the contained expression and then transfers control to the end point of the expression statement.

6.5. The *apply* statement

An *apply-statement* is used to invoke a rule.

apply-statement →

'apply' path-name '(' argument-list ')' ';' .

argument-list →

λ |

argument ',' argument-list .

argument →

ocl-expression .

For a rule invocation, the compiler must first identify the one rule to invoke or the group of overloaded rules from which to choose a specific rule to invoke. In the latter case, determination of the specific rule to invoke is based on the context provided by the types of the arguments in the *argument-list*.

The compile-time processing of a method invocation of the form *R(A)*, where *R* is a rule group and *A* is an optional *argument-list*, consists of the following steps:

- The set of candidate rules for the rule invocation is constructed. The set of rules associated with *path-name*, which are found by a name lookup operation, is reduced to those rules that are applicable with respect to the argument list *A*. The set reduction consists of applying the following rules to each rule *T::R* in the set, where *T* is the transformation in which the rule *R* is declared:
- If *R* is not applicable with respect to *A*, then *R* is removed from the set.
- If *R* is applicable with respect to *A*, then all rules declared in a base type of *T* are removed from the set.
- If the resulting set of candidate rules is empty, then no applicable methods exist, and a compile-time error occurs.
- The best rule of the set of candidate rules is identified using the overload resolution rules. If a single best rule cannot be identified, the rule invocation is ambiguous, and a compile-time error occurs.

Once a rule has been selected and validated at compile-time by the above steps, the actual run-time invocation is processed according to the rules of invocation.

6.5.1. Name lookup

A name lookup is the process whereby the meaning of a name in the context of a transformation is determined. A rule lookup may occur as part of evaluating a *simple-name* in an apply statement.

A lookup of a name *N* in a transformation *T* is processed as follows:

- The set of all accessible rules named *N* declared in *T* and the base transformations of *T* is constructed.
- If no members named *N* exist and are accessible, then the lookup produces no match.
- Otherwise, this group of rules is the result of the lookup.

6.5.2. Rule applicable to A

A rule is said to be an *applicable rule* with respect to an argument list *A* when all of the following are true:

- The number of arguments in *A* is identical to the number of parameters in the function member declaration.
- For each argument in *A*, the type of the argument is compatible to the type of the corresponding parameter, according to OCL 2.0 specification [OCL2].

6.5.2.1. Better function member

Given an argument list $A = A_1, A_2, \ldots, A_N$ with a set of argument types $T_1, T_2, \ldots, T_N$ and two applicable rules R_P and R_Q with parameter types $P_1, P_2, \ldots, P_N$ and $Q_1, Q_2, \ldots, Q_N$, R_P is defined to be a *better rule* than M_Q if

- For each argument, the implicit conversion from T_I to P_I is not worse than the implicit conversion from T_I to Q_I, and
- For at least one argument A_J, the conversion from T_J to P_J is better than the conversion from T_J to Q_J.

6.5.2.2. Better conversion

Given an implicit conversion C_1 that converts from a type S to a type T_1, and an implicit conversion C_2 that converts from a type S to a type T_2, the *better conversion* of the two conversions is determined as follows:

- If T_1 and T_2 are the same type, neither conversion is better.
- If S is T_1, C_1 is the better conversion.
- If S is T_2, C_2 is the better conversion.
- If an implicit conversion from T_1 to T_2 exists, and no implicit conversion from T_2 to T_1 exists, C_1 is the better conversion.
- If an implicit conversion from T_2 to T_1 exists, and no implicit conversion from T_1 to T_2 exists, C_2 is the better conversion.

6.5.3. Rule invocation

This section describes the process that takes place at run-time to invoke a particular rule *R*. It is assumed that a compile-time process has already determined the particular rule to invoke, possibly by applying overload resolution to a set of candidate rules.

The run-time processing of a rule member invocation consists of the following steps:

- The argument list is evaluated from left to right.
- The resulting values are used to build an activation record.
- The body of rule *R* is applied over every source model element for which the filter attached to rule *R* is true. If the source model and target model are identical, the elements added by other previous rules are discarded.

For example, the rule

```
rule r match A(self.name='John') {
     let x:B;
     x := new B;
      ...
}
```

creates a B instance for each A instance whose property *name* has the value *John*. The filter expression can be any OCL expression (e.g. navigation expressions, operation on primitive types and collections, and iterator expressions as *select* and *forall*).

6.6. The *delete* statement

A *delete-statement* destroys an object created by a *new-expression*.

delete-statement →

'delete' ocl-expression ';'.

The operand must have a model element type [OCL20].

6.7. Decision statements

Selection statements select one of a number of possible statements for execution based on the value of some expression.

selection-statement →

if-statement.

6.7.1. The *if* statement

The *if* statement selects a statement for execution based on the value of a boolean expression.

if-statement →

'iff' expression 'then' statement ['else' statement] 'endif'.

An *else* part is associated with the lexically nearest preceding *iff* that is allowed by the syntax. Thus, an *if* statement of the form

```
iff x iff y then y:= x; else x:=y;
```

is equivalent to

```
iff x then
      if y then
            y:=x;
      else
            G();
      endif
endif
```

An `if` statement is executed as follows:

- The *expression* is evaluated.
- If the expression yields *true*, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the *if* statement.
- If the expression yields *false* and if an *else* part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the *if* statement.
- If the expression yields *false* and if an *else* part is not present, control is transferred to the end point of the *if* statement.

The first embedded statement of an *if* statement is reachable if the *if* statement is reachable and the expression does not have the constant value *false*.

The second embedded statement of an *if* statement, if present, is reachable if the *if* statement is reachable and the expression does not have the constant value *true*.

The end point of an *if* statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an *if* statement with no *else* part is reachable if the *if* statement is reachable and the expression does not have the constant value *true*.

6.8. Iteration statements

Iteration statements repeatedly execute an embedded statement.

iteration-statement →

while-statement |

do-statement |

foreach-statement.

6.8.1. The *while* statement

The *while* statement conditionally executes an embedded statement zero or more times.

while-statement →

‘while’ expression ’do’ statement .

A while statement is executed as follows:

- The *expression* is evaluated.
- If the expression yields *true*, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a *continue* statement), control is transferred to the beginning of the *while* statement.
- If the expression yields *false*, control is transferred to the end point of the *while* statement.

Within the embedded statement of a *while* statement, a *break* statement may be used to transfer control to the end point of the *while* statement (thus ending iteration of the embedded statement), and a *continue* statement may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the *while* statement).

The embedded statement of a *while* statement is reachable if the *while* statement is reachable and the expression does not have the constant value *false*.

The end point of a *while* statement is reachable if at least one of the following is true:

- The *while* statement contains a reachable *break* statement that exits the *while* statement.
- The *while* statement is reachable and the expression does not have the constant value *true*.

6.8.2. The *do* statement

The *do* statement conditionally executes an embedded statement one or more times.

do-statement→

‘do’ statement ‘while’ ‘(‘ expression ‘)’ ‘;’

A do statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a *continue* statement), the *expression* is evaluated. If the expression yields *true*, control is transferred to the beginning of the *do*

statement. Otherwise, control is transferred to the end point of the *do* statement.

Within the embedded statement of a *do* statement, a *break* statement may be used to transfer control to the end point of the *do* statement (thus ending iteration of the embedded statement), and a *continue* statement may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the *do* statement).

The embedded statement of a *do* statement is reachable if the *do* statement is reachable.

The end point of a *do* statement is reachable if at least one of the following is true:

- The *do* statement contains a reachable *break* statement that exits the *do* statement.
- The end point of the embedded statement is reachable and the boolean expression does not have the constant value *true*.

6.8.3. The *foreach* statement

The *foreach* statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

foreach-statement→

'foreach' variable-declaration 'in' expression 'do' statement

The *variable-declaration* contains the declaration of the *iteration variable* of the statement. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a *foreach* statement, the iteration variable represents the collection element for which an iteration is currently being performed. The iteration variable can be modified or passed as an argument.

The type of the *expression* of a *foreach* statement must be a collection type (as defined below), and an explicit conversion must exist from the element type of the collection to the type of the iteration variable. If *expression* has the undefined value, a dynamic semantics error is reported.

A type C is said to be a *collection type* if it is declared as an OCL collection type or implements the *collection pattern* by meeting all of the following criteria:

- C is the type of a UML attribute whose multiplicity describes a set of at least 2 elements.
- C is the type of a UML association end whose multiplicity describes a set of at least 2 elements.

6.8.4. The *break* statement

The *break* statement exits the nearest enclosing *while*, *do*, or *foreach* statement.

> *break-statement →*
>
> *'break' ';'*

The target of a *break* statement is the end point of the nearest enclosing *while*, *do*, or *foreach* statement. If a *break* statement is not enclosed by a *while*, *do*, or *foreach* statement, a compile-time error occurs.

When multiple *while*, *do*, or *foreach* statement statements are nested within each other, a *break* statement applies only to the innermost statement. To transfer control across multiple nesting levels, decision statements and boolean flags must be used.

A *break* statement is executed as follows:

- Control is transferred to the target of the *break* statement.

Because a *break* statement unconditionally transfers control elsewhere, the end point of a *break* statement is never reachable.

6.8.5. The *continue* statement

The *continue* statement starts a new iteration of the nearest enclosing *while*, *do*, or *foreach* statement.

> *continue-statement →*
>
> *'continue' ';'*

The target of a *continue* statement is the end point of the embedded statement of the nearest enclosing *while*, *do*, or *foreach* statement. If a *continue* statement is not enclosed by a *while*, *do*, or *foreach* statement, a compile-time error occurs.

When multiple *while*, *do*, or *foreach* statements are nested within each other, a *continue* statement applies only to the innermost statement. To transfer control across multiple nesting levels, decision statements and boolean flags must be used.

A *continue* statement is executed as follows:

- Control is transferred to the target of the *continue* statement.

Because a *continue* statement unconditionally transfers control elsewhere, the end point of a *continue* statement is never reachable.

Chapter 7. NAMESPACES AND TRANSLATION UNITS

A YATL program consists of one or more translation units, each contained in a separate source file. When a YATL program is processed, all of the translation units are processed together. Thus, translation units can depend on each other, possibly in a circular fashion. A translation unit consists of zero or more import directives followed by zero or more declarations of namespace members: queries, views, or transformations.

The concept of namespace was introduced to allow YATL programs to solve the problem of names collision that is a vital issue for large-scale transformation systems. Namespaces are used both as an “internal” organization system for a program, and as an “external” organization system - a way of presenting program elements that are exposed to other programs. A YATL program can reuse a transformation or a query by importing the corresponding namespaces and invoking the appropriate rules.

A YATL query is an OCL expression, which is evaluated into a given context such as a package, classifier, property, or operation. The returned value can be a primitive type, model elements, collections or tuples. Queries are used to navigate across model elements and to interrogate the population stored in a given repository. YATL uses the OCL implementation that was initially developed under KMF and then under Eclipse as an open source project [OCLP].

A YATL transformation is a construct that maps a source model instance to a target model instance by matching a pattern in a source model instance and creating a collection of objects with given properties in the target model instance. The matching part is performed using the declarative features of OCL, while the creation of target instances is done using the imperative features provided by YATL. YATL provides also the possibility of interacting with the underlying machine using *native* statements. Although we do not encourage the use of such features, they were provided to support the modeller when some operations are not available at the metamodel level (e.g. the standard library of OCL 2.0 does not provide a function to convert lowercase letters to uppercase letters).

Chapter 8. CONCLUSIONS

This section contains a description of the compliance to RFP requirements, other design requirements, and related work in this area.

8.1.1. Compliance to RFP requirements

OMG's QVT RFT [QVT02] comprises a set of mandatory and optional requirements for the Query/Views/Transformations proposal. Meeting these requirements, especially the mandatory ones, is very important, because they are crucial for describing model transformations in the model driven engineering framework. This section presents these requirements and analyzes YATL's compliance with them.

8.1.1.1. Mandatory requirements

> *"1. Proposals shall define a language for querying models. The query language shall facilitate ad-hoc queries for selection and filtering of model elements, as well as for the selection of model elements that are the source of a transformation."*

YATL queries described using OCL 2.0 concepts can be used to query the source model instance. The data returned by a query can be any OCL value: number, string, boolean value, collection, tuple, or any value from the metamodel. The selection and filtering of model elements that are the source of transformation is done through the LHS of transformation rules.

> *"2. Proposals shall define a language for transformation definitions. Transformation definitions shall describe relationships between a source MOF metamodel S, and a target MOF metamodel T, which can be used to generate a target model instance conforming to T from a source model instance conforming to S. The source and target metamodels may be the same metamodel."*

The relations between source metamodel S and target metamodel T are described in YATL by translation rules with LHS and RHD. Current instances of relations can be stored so that they can be retrieved latter, using the *track* mechanism. YATL can be used to describe transformations for which the source model is identical with the target model. To avoid unnatural behavior in this particular case, the transformation engine applies the transformation rules only on the elements contained initially in the source model instance. The model elements that are added into the model instance by invoking transformation rules are not considered when the LHS of a rule is matched against the model instance.

> *"3. The abstract syntax for transformation, view and query definition languages shall be defined as MOF (version 2.0) metamodels."*

The abstract syntax of YATL is described using MOF concepts and is independent of the concrete syntax. The abstract syntax of YATL is described in Figure1. There is an ongoing research on the graphical syntax of YATL.

> *"4. The transformation definition language shall be capable of expressing all information required to generate target model from a source model automatically."*

Both the LHS and RHS of the rules are capable of expressing all the necessary information for transformations. The LSH is used to match a specific pattern against the source model instance, while the RSH is capable of describing the objects which are added into the target model instance.

> *"5. The transformation definition language shall enable the creation of a view of a metamodel."*

YATL does not support yet views. This is an area of ongoing research.

> *"6. The transformation definition language shall be declarative in order to support transformation execution with the following characteristic:*
>
> *• Incremental changes in a source model may be transformed into changes in a target model immediately."*

YATL is partially declarative, containing a mixture of declarative and imperative features. The declarative features are inherited from OCL while the imperative features are provided mainly by YATL statements.

> *"7. All mechanisms specified in Proposals shall operate on model instances of metamodels defined using MOF version 2.0."*

Both LHS and RHS of the transformation rules operate on model instances using names, pathnames, and concepts specific to the metamodels and not to their specific implementation on a given platform.

8.1.1.2. Optional requirements

> *"1. Proposals may support transformation definitions that can be executed in two directions. There are two possible approaches:*
>
> *• Transformations are defined symmetrically, in contrast to transformations that are defined from source to target.*
>
> *• Two transformation definitions are defined where one is the inverse of the other."*

The transformations described by YATL are executed in one direction, usually from source model to target model. If a reverse transformation is needed, the modeler must write that transformation by himself.

> *"2. Proposals may support traceability of transformation executions made between source and target model elements."*

The current version of YATL supports only explicit traceability of the execution, through explicit use of *track* constructions. Adding implicit traceability mechanisms is an ongoing research area.

> *"3. Proposals may support mechanisms for reusing and extending generic transformation definitions. For example: Proposals may support generic definitions of transformations between general metaclasses that are automatically valid for all specialized metaclasses. This may include the overriding of the transformations defined on base metaclasses. Another solution could be support for transformation templates or patterns."*

To support the reusability of the code YATL programs are organized in translation units and namespaces. Future versions of YATL will support abstract, overridden, and virtual transformation rules.

> *"4. Proposals may support transactional transformation definitions in which parts of a transformation definition are identified as suitable for commit or rollback during execution."*

Future versions of YATL will support transactional transformations for which all contained transformation rules are either committed or rolled back together.

> *"5. Proposals may support the use of additional data, not contained in the source model, as input to the transformation definition, in order to generate a target model. In addition proposals may allow for the definition of default values for this data."*

YATL allows the invocation of the transformation rules by passing additional data as arguments.

> *"6. Proposals may support the execution of transformation definitions where the target model is the same as the source model; i.e. allow transformation definitions to define updates to existing models. For example a transformation definition may describe how to calculate values for derived model elements."*

YATL allows the definition of transformations for which the source model is identical to the target model. For example, YATL transformations can be used to change properties' values or remove objects. To avoid unnatural behavior in this particular case, the transformation engine applies the transformation rules only on the elements contained initially in the source model instance. The model elements that are added into the model instance by invoking transformation rules are not considered when the LHS of a rule is matched against the model instance.

8.1.1.3. Issues to be discussed

> *"1. The OMG CWM specification already has a defined transformation model that is being used in data warehousing. Submitters shall discuss how their transformation specifications compare to or reuse the support of mappings in CWM."*

YATL uses the concept of repository and warehouse to store source and target model instances. These concepts are mapped into an implementation by KMF-Studio, a tool from KMF. Mapping support in CWN can easily be reformulated using YATL.

> *"2. The OMG Action Semantics specification already has a mechanism for manipulating instances of UML model elements. Submitters shall discuss how their transformation specifications compare to or reuse the capabilities of the UML Action Semantics."*

A YATL program specification can be described in terms of the Action Semantics.

> *"3. How is the execution of a transformation definition to behave when the source model is not well-formed (according to the applicable constraints?). Also should transformation definitions be able to define their own preconditions. In that case: What's the effect of them not being met? What if a transformation definition applied to a well-formed model does not produce a well-formed output model (that meets the constraints applicable to the target metamodel)?"*

YATL does not check implicitly if the source model instance or if the generated target model instance are well formed. YATL queries can be used explicitly before and after the transformation to check the pre and post conditions associated with a transformation.

> *"4. Proposals shall discuss the implications of transformations in the presence of incremental changes to the source and/or target models."*

YATL and YATL-Studio cannot automatically detect if the source or the target model instance suffered incremental changes. At this stage it is the modeler's task to keep track of the changes. In the near future, mechanisms to detect automatically if the a model instance suffered some changes will be added to the KMF warehouse and repository concepts.

8.1.2. Other design features

As well as supporting the ongoing QVT requirements, we designed YATL to support the following additional requirements:

- The syntax and semantics of YATL must be well defined.
- The process of applying the transformation rules must be deterministic.
- Queries, views, and transformations are organized in namespaces to provide reusability and avoid name collision.
- The transformation engine must be capable of performing efficient transformation for large-scale systems.
- YATL must provide enough computational expressiveness power, regardless of the host platform or language. For example, YATL should support a complete set of operations on basic types like strings, integers, or floating point numbers.

8.1.3. Relationship to existing OMG specifications

Object Constraint Language OCL forms the basis of the query language and is also used to match the LHS of the transformation rules.

Meta Object Facility The abstract syntax of YATL and OCL is described in terms of MOF; the superstructure is a slightly more involved extension of MOF.

Common Warehouse Metamodel Concepts like warehouse and repository are used to store source and target model instances.

8.1.4. Comparison to QVT submissions

Since OMG launched its QVT RFP [QVT02] in 2002, several submissions were made. DSTC's submission [QVTD] contains a declarative definition of QVT and uses high-level concepts that are similar with those from Prolog. Unfortunately it cannot cope with large-scale transformations because its concepts make the implementation very slow. QVT Partners submission [QVTP] considers that transformations are special cases of relations and describes them using a graphical syntax. This approach is similar to the one presented in [ASP03]. This submission provides a mechanism for relations' refinement. In the near future YATL will provide a similar support, although it will be described in a textual way. The French submission [QVTF] is very similar to the approach that we took. But, there are a lot of differences such as the concrete syntax, the semantics of the rules, the tracking mechanism, the support for interaction with the host machine and creation of target model instance.

Appendix 1. GRAMMAR SPECIFICATION RULES

Grammar specification is done using the following rules:

1) Left hand-side and right hand-side are separated by symbol $\rightarrow$.

2) Each production ends with a dot.

3) Terminal symbols are written using capital letter or delimited by apostrophes.

4) The following shortcuts are permitted:

Shortcut	Meaning
$X \rightarrow \alpha\ (\ \beta\ )\ \gamma\ .$	$X \rightarrow \alpha\ Y\ \gamma\ .\ Y \rightarrow \beta\ .$
$X \rightarrow \alpha\ [\ \beta\ ]\ \gamma\ .$	$X \rightarrow \alpha\ \gamma \mid \alpha\ (\ \beta\ )\ \gamma\ .$
$X \rightarrow \alpha\ u + \gamma\ .$	$X \rightarrow \alpha\ Y\ \gamma\ .\ Y \rightarrow u \mid u\ Y\ .$
$X \rightarrow \alpha\ u * \gamma\ .$	$X \rightarrow \alpha\ Y\ \gamma\ .\ Y \rightarrow u \mid u\ Y \mid \lambda\ .$
$X \rightarrow \alpha \parallel a.$	$X \rightarrow \alpha\ (\ a\ \alpha\ )\ *\ .$

where α, β and γ are strings over the language alphabet, Y is a symbol which does not appear elsewhere in the specification, u is either a unique symbol or an expression delimited by parentheses, and a is a terminal symbol.

Appendix 2. YATL-LEXICAL GRAMMAR

Five basic elements make up the lexical structure of a YATL source file: line terminators, white space, comments, and tokens. Of these basic elements, only tokens are significant in the syntactic grammar of a YATL program.

For compatibility with source code editing tools that add end-of-file markers, and to enable a source file to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to every source file in a C# program:

- If the last character of the source file is a Control-Z character, this character is deleted.
- A carriage-return character is added to the end of the source file if that source file is non-empty and if the last character of the source file is not a carriage return, a line feed, a line separator, or a paragraph separator.

The *input* production defines the lexical structure of a YATL source file. Each source file in a YATL program must conform to this lexical grammar production.

input $\rightarrow$ λ | *input-element* | *input input-element.*

input-element $\rightarrow$ *line-terminator* | *whitespace*| *comment*| *token.*

Line terminators divide the characters of a C# source file into lines. YATL uses the following markers to indicate the end of a line:

- Carriage return character (*U+000D*)
- Line feed character (*U+000A*)
- Carriage return character (*U+000D*) followed by line feed character (*U+000A*)
- Next line character (*U+0085*)
- Line separator character (*U+2028*)
- Paragraph separator character (*U+2029*)

YATL's tokens are based on OCL tokens [OCL20],[ALP03]. It adds only the following keywords:

apply	do	namespace	start
break	foreach	new	track
build	import	null	transformation
continue	in	query	while
delete	match	rule	

and the assignment operator :=.

Appendix 3. YATL-Syntax Grammar

translation-unit →

import-list starting-rule namespace-declaration-list .

import-list →

λ |

import-list import-declaration .

import-declaration →

'import' simple-name '.' '' ';' .*

starting-rule →

'start' pathname ';' .

namespace-declaration-list →

λ |

namespace-declaration-list namespace-declaration .

namespace-declaration →

'namespace' simple-name '(' models ')' '{' (query|transformation) '}' .*

models →

source-model [',' target-model].

transformation →

'transformation' simple-name '{' rule '}' .*

rule →

'rule' simple-name filter '(' [param (',' param)] ')' compound-stm .*

filter →

'match' filter-path .

filterPath →

filter-step |

filter-path '::' filter-step .

filter-step →

simple-name ['[' ocl-expression ']']

statement-list →

λ |

statement-list statement .

statement →

declaration-stm |

expression-stm |

compound-stm |

if-stm |

loop-stm |

break-stm |

continue-stm |

apply-stm .

declaration-stm →

'let' variable-declaration-list ';' .

expression-stm →

[expression ';'] .

compound-stm →

'{' statement-list:list '}'.

if-stm →

'iff' ocl-expression 'then' statement ['else' statement] 'endif'.

loop-stm →

'while' ocl-expression 'do' statement |

'do' statement 'while' '(' ocl-expression ')' ';' |

'foreach' variable-declaration 'in' ocl-expression 'do' statement .

break-stm →

'break' ';' .

continue-stm →

'continue' ';' .

apply-stm →

'apply' pathname '(' [ocl-expression (',' ocl-expression)] ')' ';'*

delete-stm →

'delete' ocl-expression ';' .

expression →

assignment-expression |

ocl-expression |

track-expression .

assignment-expression →

ocl-expression ':=' rhs-expression .

rhs-expression →

ocl-expression |

new-expression |

build-expression |

track-expression .

new-expression →

'new' path-name .

build-expression →

'build' path-name '{' [pair (',' pair)] '}'*.

pair →

name ':=' rhs-expression .

track-expression →

'track' '(' ocl-expression ',' simple-name ',' ocl-expression ')' |

'track' '(' 'null' ',' simple-name ',' ocl-expression ')' |

‘track’ ‘(‘ ocl-expression ‘,’ simple-name ‘,’ ‘null’ ‘)’ .

query →

‘query’ simple-name ‘{‘ context-declaration-list ‘}’ .

Nonterminal ocl-expression, variable-declaration, and context-declaration-list are described in [OCL2] and [ALP03].

BIBLIOGRAPHY

[AP03] Akehurst D. and Patrascoiu O. (2003). OCL 2.0 – Implementing the Standard for Multiple Metamodels. In *OCL2.0-"Industry standard or scientific playground?" - Proceedings of the UML'03 workshop*, page 19. Electronic Notes in Theoretical Computer Science.

[ALP03] Akehurst D., Linington P., and Patrascoiu O. (2003). OCL 2.0 – Implementing the Standard. Technical Report No. 12-03, Computer Laboratory, University of Kent, UK.

[AKP03] Akehurst D., Kent S., and Patrascoiu O. (2003). A relational approach to defining and implementing transformations between metamodels. In *Journal of Software and Systems Modeling* (SoSym), 2(4), 215-239.

[CH03] Czarnecki K., and Helsen S. (2003). Classification of Model Transformation Approaches. In *Generative techniques in the context of MDA – Proceedings of OOPSLA 2003 workshop*.

[CWM] OMG, Common Warehouse Metamodel Specification. OMG Document formal/2003-03-02, available at http://www.omg.org/cwm.

[EMF] IBM, Eclipse Modeling Framework. http://www.eclipse.org.

[Fra03] Frankel D. S. (2003) *Model Driven Architrecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[GLRSW02] Gerber A., Lawley M., Raymond K., Steel J., and Wood A. (2002). Transformation: The Missing Link of MDA, in A. Corradini, H. Ehring, H. J. Kreowsky, G. Rozenberg (Eds): In *Proc. of Graph Transformation: First International Conference (ICGT 2002)*

[GHK99] Gil J., Howse J, and Kent S. (1999) Formalising Spider Diagrams, In *Proc.of IEEE Symposium on Visual Languages (VL99)*, IEEE Press, 130-137.

[Java] Java standard http://www.sun.com

[KMF] Kent Modeling Framework. http://www.cs.kent.ac.uk/projects/kmf.

[MDA] MDA. Model Driven Architecture Specification. OMG document omg/03-06-01, available at. http://www.omg.org/mda.

[MOF] OMG, MOF Meta Object Facility Specification, OMG Document formal/2002-04-03, available at http://www.omg.org/mof

[OCL] OMG, OCL Object Constraint Language Specification Revised Submission, Version 1.6, January 6, 2003, OMG document ad/2003-01-07.

[OCL2P] OCL Open source project: Object Constraint Language for Kent Modeling Framework and Eclipse Framework. http://www.cs.kent.ac.uk/projects/kmf.

[OMG] OMG Object Management Group. http://www.omg.org.

[UML] OMG, Unified Modeling Language Specification, Version 1.5, 2003, OMG Document formal/2003-03-01, available at. http://www.omg.org/uml.

[QVT02] OMG, QVT Query/Views/Transformations RFP, OMG Document ad/02-04-10, revised on April 24, 202. http://www.omg.org/cgi-bin/doc?ad/2002-4-10

[QVTD] OMG, MOF Query/Views/Transformation, Initial submission, DSTC and IBM.

[QVTP] OG, MOF Query/Views/Transformation, Initial submission, QVT Partners.

[QVTF] OMG, MOF Query/Views/Transformation, Initial submission, Alcatel, SoftTeam, Thales, TNI-Valiosys.

[Pat04] Patrascoiu O. (2004) YATL:Yet Another Transformation Language. In Proc. of First European Workshop MDA-IA, University of Twente, the Nederlands.

[RJB99] Rumbaugh, J., Jacobson I., and Booch G.. (1999). The Unified Modeling Language – Reference Manual. Addison-Wesley.

[WK99] Warmer, J. and Kleppe A. (1999). The Object Constraint Language: Precise Modeling with UML. Addison-Wesley.

[UNI] Unicode standard. http://www.unicode.org

[XMI] OMG, MOF Meta Object Facility Specification OMG Document 2003-05-02, available at http://www.omg.org/uml