T U M

I N S T I T U T F Ü R I N F O R M A T I K

The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA

Bernhard Schaetz

TUM-I0111
Dezember 01

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-12-I0111-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2001

Druck: Institut für Informatik der
Technischen Universität München

The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA

Bernhard Schätz, Fakultät für Informatik, TU München
Email: schaetz@in.tum.de

Abstract: Using a model-based development process for embedded software systems, the process of developing descriptions of systems can be interpreted as performing operations on the model of the system. To increase efficiency of such a development process, general high-level operations can be defined manipulating these models, for instance copying a component including its behavior or refactoring a state-based description. In this report we introduce a language for the description of such operations on models called ODL (operation description language). Furthermore, a framework is introduced for the evaluation of ODL expressions.

Keywords: Model-based, constraint language, meta model, evaluation, modification, AutoFocus, formal methods, description techniques, conceptual model, high-level operation, application

CR-Classification: D.2.2 Tools and Techniques, D.2.4 Program Verification, D.2.10 Design, D.2.m Miscellaneous, D.3.1 Formal Definitions and Theory, F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.1 Mathematical Logic

1 Goal

Goal of the framework and language definition is the introduction of a language supporting the definition of operations on the AutoFocus/Quest meta model independently of an actual implementation of the meta model. By means of those operations a model-based development process is supported by high-level consistency checks as well as modification of the model of the system. The language is designed to be as operational as possible while maintaining an abstract logical flavor for both efficient and high-level descriptions of model operations. The abstract flavor of the language is needed to support reasoning about the properties of operations defined using ODL. This is important since operations are expected to manipulate the model in a well-defined manner. Thus, by making use of the restricted possibilities of manipulations of ODL operations, the correctness of the operations with respect to those well-definedness conditions can be proved statically more easily than using – for example – fully-fledged Java for the descriptions of operations. The verification of those well-definedness conditions is out of the scope of this report and is not treated here. See, e.g., [Sch00] for a discussion on how to verify properties of a constraint language.

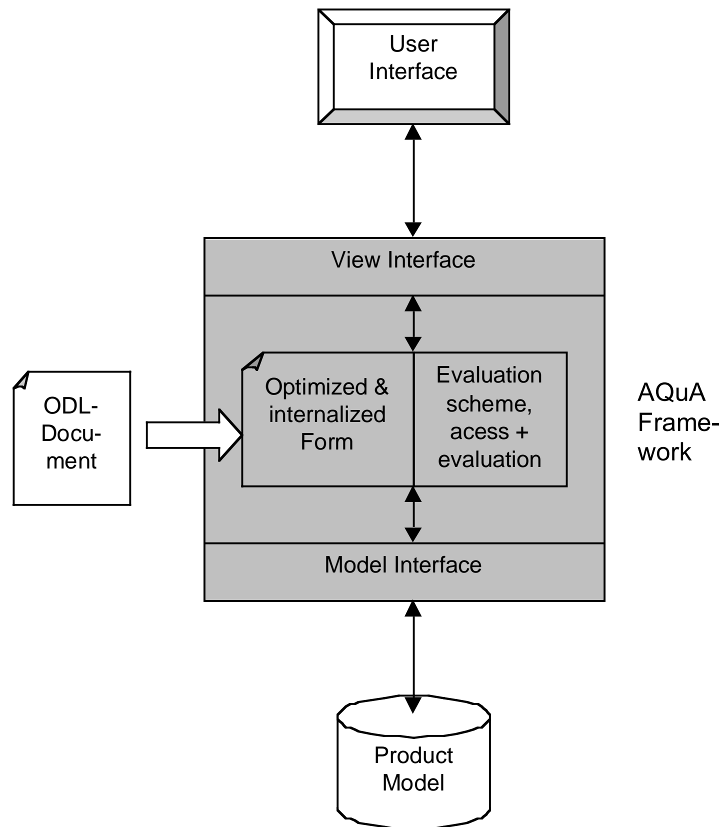

Figure 1: System Structure

The overall structure of the system is shown in Figure 1. To create a new operation, a ODL description of the operation is defined. AQuA parses and internalizes the definition using the parse tree, generating an evaluation tree and an appropriate evaluation scheme described in section 6. As the internal form represents the definition in an executable way (an object tree with nodes accessing the model, the user interface or querying other nodes), the operation becomes part of the operations of AutoFocus/Quest.

For a description of the AutoFocus modeling techniques, the definition of consistency conditions and the model based process see [HS01].

1.1 Meta Model

Basically, the meta model describes, which modeling concepts are used during the development concepts (like components, channels, ports, states, transitions) and how those concepts are related (for instance, "a channel connects two ports"). Depending on the domain the meta model is used for, different elements of a meta model might be needed (for example, data oriented concepts like class, object, association, as found in UML [FS97]; capsules, history states and inter-level transitions as found in ROOM [SGW94]). In the following the basic meta model of AutoFocus/Quest is used; a simplified version is shown in Figure 2. A more complete version of an AutoFocus/Quest meta model is shown in Figure 15 and explained in more detail in [BL+00].

For the definition of ODL, the meta model is defined in the following a way.

A meta model *MM* consists of a pair *(ME,MR)* of a family of meta model entities $ME = \{ME_1,...ME_m\}$ and a family of meta model relations $MR = \{MR_1,...,MR_n\}$. Each

meta model relation ME_i is a relation of meta model entities of the form $MR_i \subseteq ME_{j_1} \times \ldots \times ME_{j_k}$ with $k > 0$. Meta model entities are pairwise disjunct sets of model elements – making model elements instances of meta model elements. For example, in the meta model mentioned above, meta model elements are components, ports, states, etc. with all possible instances as their set of model elements.

For the simplicity of the model, attributes of model elements are introduced as a canonic shorthand formalization of special meta model relations. If a meta model entity *E* has an attribute *att* of type *type*, this is interpreted as the existence of a meta model relation RE_{att} of type $E \times type$.

In ODL, meta model elements are the types variables are quantified over using forall, exisis, context, or new. The attribute *att* of an entity *e* is noted as *e.att*. Relation instances of a meta model relation $R \subseteq E_1 \times \ldots \times E_k$ between entities $e_i \in E_{j_i}$ are noted as $R(e_1,\ldots,e_k)$.

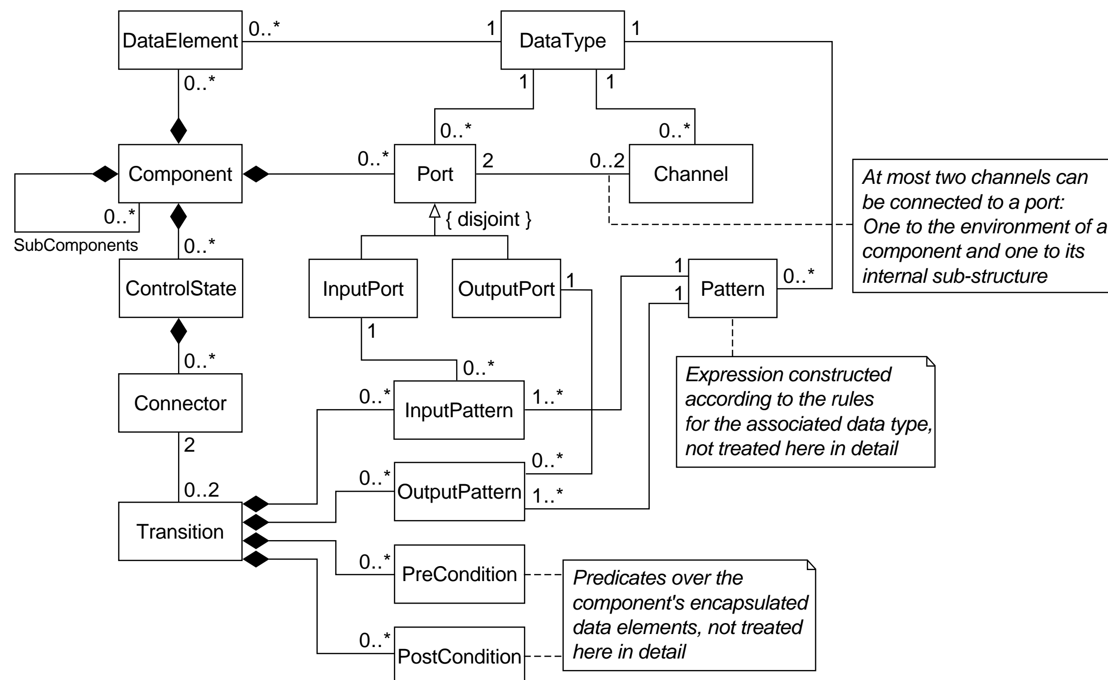

Figure 2: Simplified AutoFocus/Quest Meta Model

1.2 Product Model

If the meta model is compared with an E/R-diagram/database scheme or a class diagram, the product model can be compared to a concrete instance of those like a database or an object diagram. Thus, while operations like SQL queries or ODL expression are defined using the E/R-diagram/database scheme or the meta model, resp., the operations are carried out using the database instance or the product model, changing the state of it.

2 Examples

Examples for the application of the query and update language:

- Renaming: Find and replace names of specification objects: As an example we will replace all occurrences of (the name of) a port variable „old“ by „new“ in a SSD-component:

```
context old_name : String.  /* Old name to be replaced */
    context new_name : String.  /* New name to be replaced with */
        context comp: Component. /* Component to replace within */
            forall port: Port. /* All ports of component */
                (is_port(comp,port) and is_name(port,oldname)
                            implies (result not is_name(port,oldname and
                            result is_name(port,oldname))
```

  Assigning “old” to `old_name` and “new” to `new_name` and the component to `comp`, the evaluation of the ODL-expression performs the corresponding modification.

- Query for the validity of consistency conditions. See [HS01] for examples of consistency conditions.
- Replace a specification object by a complex object. See [HS99] for an example like the application of specification modules.

3 Basics of the Language

The definition of the AQuA language is based on the concepts of predicate logic. An operation is defined via a corresponding predicate using universal and existential quantification as well as logical junctors like conjunction, disjunction, negation and implication. Thus, in a simplified version, the grammar could be basically defined

<Formula>	Forall <Variable>:<Type> . <Formula>
<Formula>	Exists <Variable>:<Type> . <Formula>
<Formula>	<Formula> and <Formula>
<Formula>	<Formula> or <Formula>
<Formula>	<Formula> implies <Formula>
<Formula>	(<Formula>)
<Formula>	not <Formula>

Furthermore, appropriate production rules for the definition of basic predicates are needed. However, predicate calculus must be extended for two reasons:

1) Updates cannot be expressed in classical calculus since it does not deal with before/after situations.
2) Predicate calculus offers no means of user interaction while operations on the meta model generally require user input.

To deal with these limitations, two extensions are introduced:

- A new quantor “context” which is treated similarly to a standard existential quantor. Operationally, the variable introduced by this quantor gets its value assigned by the user.
- For updates, two operators are introduced:
  - A quantor “new” for the introduction of new entities, again similar to a standard existential quantor.
  - A new operator „result“ of arity one to modify relations (add or delete relations).

Thus, the basic grammar is extend by

<Formula>	context <Variable>:<Type> . <Formula>
<Formula>	new <Variable>:<Type> . <Formula>
<Formula>	Result <Formula>

This is only a very reduced form of the grammar, being to liberal concerning result, among other things. For a more complete description of the Operation Definition Language (ODL) see the appendix 9.1.

4 Basic Sorts, Clauses and Meta Model

ODL allows the formulation of basic query/update operations. However, to evaluate the formulas, a connection between the basic clauses and the meta model must be given. The information coded in the meta model is coded by defining corresponding basic sorts predicates.

Basic sorts are all sorts occurring as classes in the meta model, i.e.

- Basic classes like Integer, Boolean, String
- Conceptual classes like component, port, channel, state, transition,

Basis clauses are all clauses for which a corresponding relation (association) exists within the meta model:

- Basic associations induced by attributes of components, ports, etc. like name, type, etc.
- Conceptual associations like “is_input_port_of(channel)”, “is_sub_state_of(state)”.

To check for the existence of a certain object or association, simply the existence of an appropriate individual or the validity of a corresponding clause is checked (see sections 6.3.3, 6.3.5, or 9.3). To create an object or to establish a association, the new-quantor or the result operator is used. To eliminate an association, the result-operator and the negation are used.

5 Extensions

So far, ODL does not support the definition of named predicates and thus the definition of recursive predicates via a fixed point operator. As an extension, section

9.2 introduced a more sophisticated type system increasing the expressive power of ODL.

6 Evaluation Algorithm For Basic AQuA (CCL)

In the following subsections we will describe the algorithm used for the evaluation of ODL terms. For reasons of simplicity we will first describe only the part of the language used for the definition of consistency conditions This part of the language, called the CCL-subset (Consistency Constraint Language) essentially can be evaluated long the lines of algorithms for first order predicate calculi over finite domains. Based on this algorithm we will then introduced the extensions:

- Result operators (new/result) to modify the model
- Extended type concept (products, sets, recursive sets)

These extensions can be essentially achieved by minor changes of the introduced data structures; the basic strategy and architecture of the algorithm remain unchanged

6.1 Basic Idea

The algorithm basically evaluates a given closed first-order predicate calculus expression with finite universe over typed variables and checks whether it meets a given Boolean value. Additionally, the set of satisfying assignments for the quantified variables of the expression is constructed. To check the validity of an expression and to generate counter examples it thus suffices to test the expression for *false*.

Note that the algorithm can be extended to open expressions using an initial assignment of free variables. This approach was used for the definition of document-based consistency conditions as described in [HSE97].

6.2 Associated Data

Two forms of information are needed for the evaluation of the expression. This information is constructed for each sub expression of the complete expression during the evaluation:

- Assignment of free variables: The mapping of variables not bound or bound by quantors “outside” of the sub expression to values of the universe.
- Evaluation: The evaluation of the (sub) expression under consideration using the associated assignment of free variables.

For the construction of suitable assignments of the bound variables of the (sub) expression leading to the Boolean value tested for the following forms of information are needed:

- Goal value: The value the (sub) expression under consideration is tested for using the associated assignment of free variables and a possible assignment of bound variables.
- Set of assignments: A set of assignment for bound variables of the (sub) expression under consideration leading to the goal value as evaluation.

An efficient implementation of this forms of information is described in section 6.4.

The evaluation algorithm is defined as a run through the internal representation of the expression. In the following, the run is described as an attribute flow graph. For each node of the graph of the internal representation we describe the associated attributes and the construction of their values. For the top-down flow, the following attributes are used:

- Free: The assignment of the free variables of the sub expression.
- Goal: The goal value the sub expression is tested for.

For the bottom-up attribute flow the following attributes are needed:

- Value: The value the sub expression evaluates to using the assignment of free and bound variables.
- Bound: The[1] set of assignments of all variables bound within the sub expression under consideration leading to the value of the sub expression.

The internal representation and the evaluation scheme can be described by the following object graph:

- Node of evaluation: Super class with the attributes as defined above supporting the basic recursive evaluation scheme by querying son nodes. All further classes of nodes are inherited form this class.
  - Attributes: Free: Assignment (list of variable/value associations), Goal: Boolean, Value: Boolean, Bound: Set of assignments
  - Methods: Evaluate: Construct Value and Bound given Free and Goal.
- Quantor node: Extends the evaluation node by executing a repeated construction of the assignment of free variables and the recursive evaluation of the node of the sub expression; classed of the existential and universal nodes are inherited form this class.
  - Attributes: additionally: Identifier: Identifier of the quantified variables, used as index for the Free assignment
  - Methods: no additional methods
- Operator node: Extends the evaluation node by the combination of the evaluations of the son nodes; one sub class per kind of operator (conjunction, disjunction, implication, equivalence, negation); no further attributes or methods.
- Basic node: Extends the evaluation node by the evaluation of a constant expression with respect to the product model; since we have equality as a relation together with the basic relations of the conceptual model, a separate class for each relation is needed.
  - Attributes: additionally :Identifier of the relation of the product model or reference to the associated object to evaluate the constant expression using the product model.
  - Methods: no additional method

The functionality of these classes of nodes is described in the following subsections. For purpose of simplicity the evaluation algorithm is described using a notation close to abstract syntax tree diagrams rather then object diagrams.

To extend the language towards complete ODL further classes of nodes are needed:

[1] If not all appropriate assignments are constructed, for example due to efficiency reasons, the resulting set if not uniquely defined.

- Context quantor nodes: Assignment of a variable by user interaction; might be generalized together with the quantor nodes by a common super class.
- Generation node: Similar to the existential quantor node; also changes the assignment of free variables; might be generalized by a common super calls together with quantor nodes and context quantor nodes.
- Modification node: Modification of the product model by insertion and deletion or basic relations; has a subclass for each of these operations; similar to basic nodes.

The functionality of these nodes is described in detail in section 7.

Furthermore, node classes are needed for the entities, too. These classes depend on the type concept used in the language and are defined in detail in sub section 9.2.

6.3 Operations

In the following sub sections the algorithm is illustrated using three constructors for expressions of the reduced first order predicate calculus:

- Negation
- Disjunction
- Universal quantification

The algortihm works similar for the further constructors of the language. In the following subsection, both the basical functionality of the evaluation and the improvement of its efficiency is discussed.

Finally, the evaluation of

- Basic Predicates

is defined; here, we will demonstrate the evaluation algorithm for binary predicates like the equality of individuals.

6.3.1 Negation

The simplest form of evaluation is applied in case of negation. Here, only trivial operations are executed:

- The assignment of the free variables of the negated expression is the assignment of the negating expression.
- The goal value of the negated expression is the negation of the goal of the negating expression.
- The result value of the negating expression is the negation of the result value of the negated expression.
- The set of assignments of bound variables of the negating expression is the set of assignments of the negated expression.

Figure 3 shows the evaluation of the attributes values for the negation.

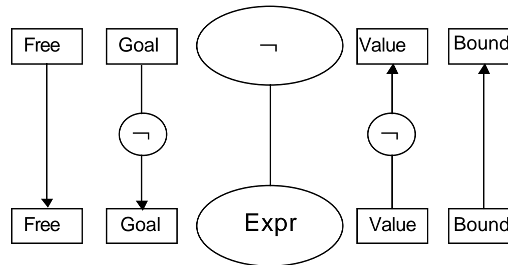

Figure 3: Negation

Due to constant complexity an improvement of efficiency is not possible

6.3.2 Disjunction

The evaluation algorithm is of the same simplicity in case of disjunction:

- The assignment of the free variables of the disjuncted expressions is the assignment of the free variables of the disjuncting expression.
- The goal value of the disjuncted expressions is the goal value of the disjuncting expression.
- The result value of the disjuncting expression is the disjunction of the result values of the disjuncted expressions.
- The set of assignments of the disjuncting expression is the union of the assignments of the bound variables of the disjuncted expressions.

Remark: Note that the set of assignments of the bound variables of the disjuncted expressions may each be defined over a different set of variable identifiers if different quantor constructions are used in the sub expressions. Thus, the resulting set contains assignments defined over different domains. This, however, does not pose a problem to the algorithm.

Note: In case of conjunction, the product of the sets of assignments of the sets of the conjuncted expressions is needed. For a coding of the assignment sets supporting an efficient construction of the set product see subsection 6.4.

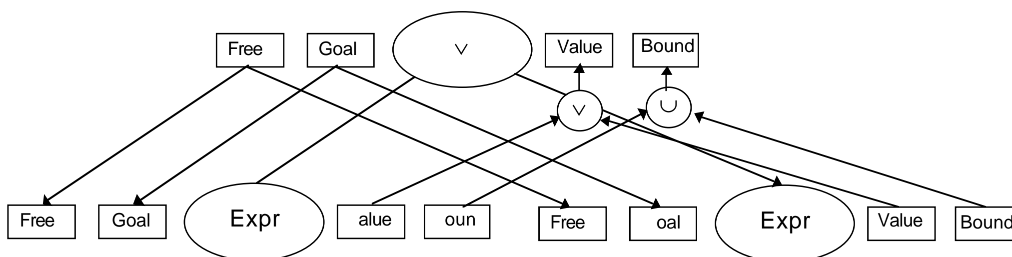

Figure 4: Disjunction

Since, again, the algorithm is of constant complexity, no improvement of efficiency is possible.

6.3.3 Universal Quantification

The most complex evaluation algorithm is needed in case of quantification, for instance universal quantification. In this subsection we describe an algorithm that performs evaluation by iteration over the evaluation for the qualified expression. Thus, several runs for the sub expression are needed. Alternatively, a set-based

construction can be used using a set-based reformulation of the above introduced evaluation algorithms. The basic complexity of both variants is identical.

- The assignment of the free variables of the quantified expression is the assignment of the free variables of the quantifying expression with addition of the corresponding assignment of the qualified variable for each iteration.
- The goal value of the quantified expression is the goal value of the quantifying expression.
- The result value of the quantifying expression is the conjunction of the result values of the quantified expression of each iteration.
- The assignment of the bound variables of the quantifying expression is the set of assignments of the bound variables with the addition of the assignment of the quantified variable for each iteration with the corresponding result value (i.e. the union of all bound variables of the quantified expression for each instantiation of the quantified variable).

Figure 5: Universal Quantification

Improvement of efficiency: In case of universal quantification the following improvement of efficiency is possible:

- If the goal value is True while the result value for an instantiation is False, the evaluation can be aborted with result value False and an empty assignment.

Note: There is no algorithmic difference concerning the assignment of bound variables between the universal and existential quantification unless efficiency improvements are considered. In both cases all assignments of the quantified variable are added in case of an adequate evaluation of the quantified sub expression. The intuitive interpretation of the bound variables may however by more suggestive in case of existential quantification.

6.3.4 Context Quantification

As described above, the context quantor is interpreted similarly to an existential quantor. Thus, the assignment of the free variables must be adapted analogously. Evaluation, however, is performed differently: the context quantor is only evaluated once and not iterated as in the case of the existential and universal quantor. For the assignment of the quantified variable, the user is inquired for an appropriate value.

- The assignment of the free variables of the quantified expression is the assignment of the free variables of the quantifying expression with addition of the corresponding assignment of the qualified variable

- The goal value of the quantified expression is the goal value of the quantifying expression.
- The result value of the quantifying expression is the conjunction of the result values of the quantified expression of each iteration.
- The assignment of the bound variables of the quantifying expression is the set of assignments of the bound variables with the addition of the assignment of the quantified variable if evaluation yielded the corresponding result value.

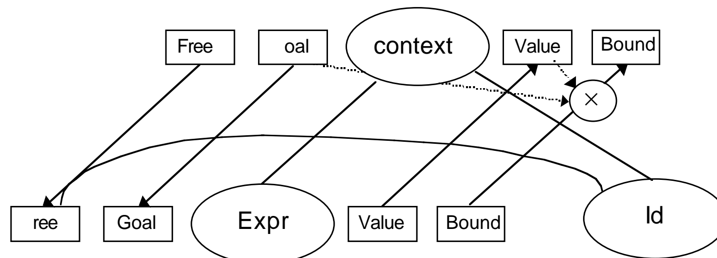

Figure 6: Context Quantor

6.3.5 Base Predicate

Similar to universal, existential and context quantification the application of a base predicate relates the evaluation to the product model in question. Unlike the quantors, however, the base predicate is interpreted as a leave node of the evaluation tree. Thus, only the flow of the attribute values for bottom-up attributes must be defined:

- The result value of the base predicate is True if such an instance of the named relation with the assignment of the free variables exists; otherwise, false.
- The set of assignments of the bound variables of the base predicate is the assignments of the free variables if result and goal value are equal, else empty.

Figure 7 shows the evaluation in case of a base predicate.

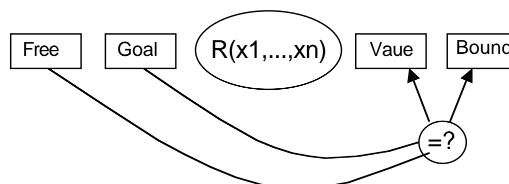

Figure 7: Base Predicate

6.4 Efficient Coding of Assignments

To encode the sets of assignments efficiently the following form similar to multi-valued decision diagrams can be used due to the kind of construction operations used:

- The set of assignments is represented as a tree (directed, acyclic graph with a distinguished root element, however not necessarily with unique access paths[2]) with both node- and edge marks.
- Internal nodes of the tree are marked with a variable identifier.
- Terminal nodes (leaves) are unmarked.

[2] For reasons of efficiency we allow different paths from the root leading to the same internal node or leave.

- The edges are marked with possible assignments of variables associated with preceding nodes.

Figure 8 shows a trivial tree of depth 1.

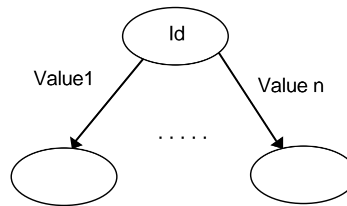

Figure 8: Assignment Tree

Using this encoding, an assignment is a path from the root to a leave: the set of variable/value-pairs is the collection of nodes/succeeding edges of this path. The set of assignments is the set of all such path-sets in the tree.

To encode the product of two assignments, the leaves of one tree are substituted by the root of the second tree. The resulting tree is again an assignment tree and encodes the set of the products of assignments.[3] For improvement of efficient the second tree is not copied but each leave is substitute with the same root. This is possible since we use the generalized version of a tree without unique paths.

To encode the union of two assignments starting with the same root node, simply the root nodes are identified and thus the union of the edges is constructed. To increase space efficiency, trees of the same assignment structure can be simplified by eliminating branchings leading to the same node.

7 Model Modification

As described above, ODL extends CCL by offering constructs to describe the modification of the model. To describe those modifications, new attributes must be added to the evaluation algorithm. In the following subsections we will introduce the new attributes and the necessary extensions of the algorithm.

7.1 Attributes

Since the modifying operators new and result only influence the result of the evaluation algorithm, the associated attributes are only computed bottom-up. The following modification operations are available:

- Introduction of a new entity
- Deletion of an existing relation
- Creation of a new relation

Correspondingly, three new attributes are needed for the extended evaluation algorithm:

- New: the set of entities introduced during the evaluation of the expression
- Add: the set of relations introduced during the evaluation of the expression.

[3] Here, we assume the uniqueness of variable identfiers along a path.

- Del: the set of relations deleted during the evaluation of the expression.

7.2 Extended Evaluation Algorithm

7.2.1 Introduction of entities

As described above, the operator for the introduction of new entities is interpreted similarly to an existential quantor. Thus, during the evaluation of the operator, the assignment of the free variables must be adjusted.

- The set of assignments of the free variables of the new-quantified expression is the set of assignments of the free variables of the quantifying expression extended with the assignment of the newly generated entity. Note that all quantifications over variables of this type of entity within this expression are not influenced by the creation of the entity. This is due to the fact that those quantors are related with the model prior to the operation, the generating quantor however with the model after the execution of the operation.
- The goal value of the quantified expression is the goal value of the quantifying expression.
- The result value of the quantifying expression is the result value of the quantified expression.
- The set of assignments of the bound variables of the quantifying expression is the set of the assignments of the bound variables of the quantified expression extended with the assignment of the quantified variables if goal and result value coincide.
- The set of the generated entities of the quantifying expression is the set of the generated entities of the quantified expression extended with the newly generated entity.
- The set of added relations of the quantifying expression is the set of the added relations of the quantified expression.
- The set of deleted relations of the quantifying expression is the set of the deleted relations of the quantified expression.

Figure 9 shows the evaluation of the attributes in case of the new operator.

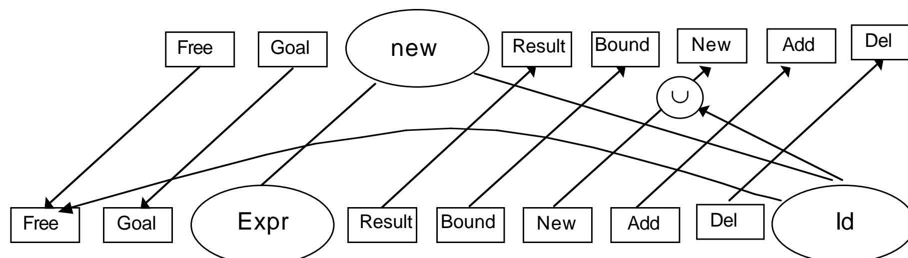

Figure 9: New Operator

7.2.2 Modification of Relations

Like with base predicates, the modification of a relation is interpreted as a leaf node that links the evaluation with the model of the system. However, unlike a base predicate, modification by definition always succeeds and thus is always evaluated to *true.* Thus, the result value of the expression is always *true*:

- The result value of the modification expression is true.

- The set of assignments of the bound variables of the expression is the set of the free variables of the expression if goal and result value coincide. Otherwise, the set is empty.
- The set of added entities as well as the set of deleted relations of the expression is empty.
- The set of added relations of the expression is the set containing the newly created relation with the variables assigned as defined by the set of assignments of the free variables of the expression.

Figure 10 shows the evaluations of the attributes in case of the modification of a relation.

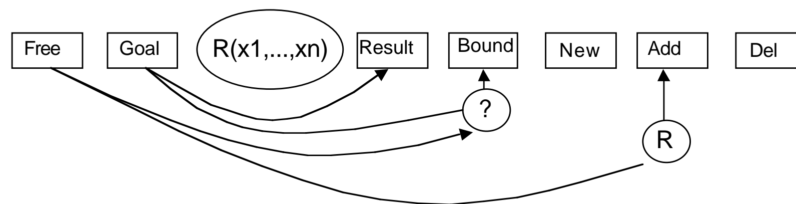

Figure 10:Modification of a relation

8 Optimization

In this section we describe the process transforming an evaluation structure (corresponding to a reduced form of an attributed abstract syntax tree) into an optimized version. The structure (actually a tree structure) is optimized considering processing-time. Here, we will only describe a simple form of optimization considering the evaluation of existential quantors and context quantors.

8.1 Basic Approach

The optimization is essentially performed by restructuring the evaluation tree. Here, the context quantor is placed ahead of surrounding existential quantors. Thus it is possible to avoid a dynamic evaluation of the assignment environment of the context quantor.

For this form of optimization the basic concepts of ODL suffice. For further optimizations advanced type concepts including set comprehension are needed.

8.2 ODL-Evaluation Tree

The ODL-syntax (as described in the appendix) essentially corresponds to a syntax of first order predicate logic with type variables (including complex types as records; see appendix). For the internal form an evaluation tree consisting of objects as described in section 9.3 is used to obtain a form freed from syntax elements as much as possible; this supports an improved evaluation as well as an improved optimization.

8.3 Optimization Rules

In the following subsection the optimization rules transforming the evaluation structure are described.

8.3.1 Existential Optimization

Goal of this optimization is the repeated evaluation of context quantors in the context of existential quantors[4] To avoid reevaluation of a context quantor in the context of a changed assignment of a universally quantified variable we can either check changes in the value of the Free attribute as described below or use a static optimization strategy. For this static variant, optimizing runtime, context quantor nodes are move "upwards" in the tree of the internal representation of the expression, as shown in Figure 11.

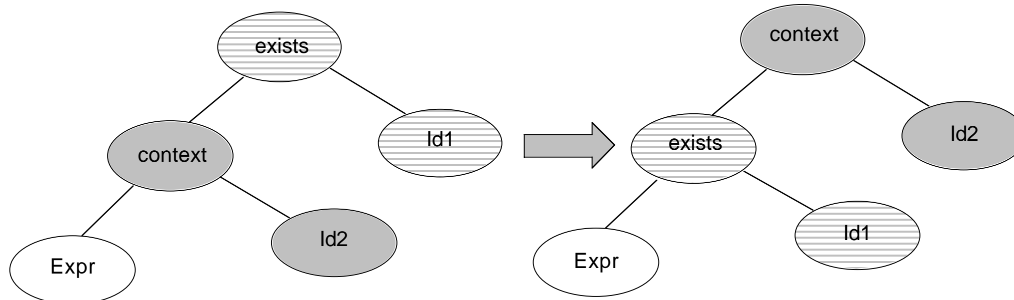

Figure 11: Context Optimization

Figure 12 shows the transformation to optimize disjunction. The transformation can be defined analogously for the other operands of arity one or two (negation, conjunction, implication, equivalence) as well as for the symmetric case with the quantor in the right branch of the tree. . Since the result operator, as introduced in the grammar in section 9.1, can only be applied to basic relations, an optimization concerning this operator is not necessary.

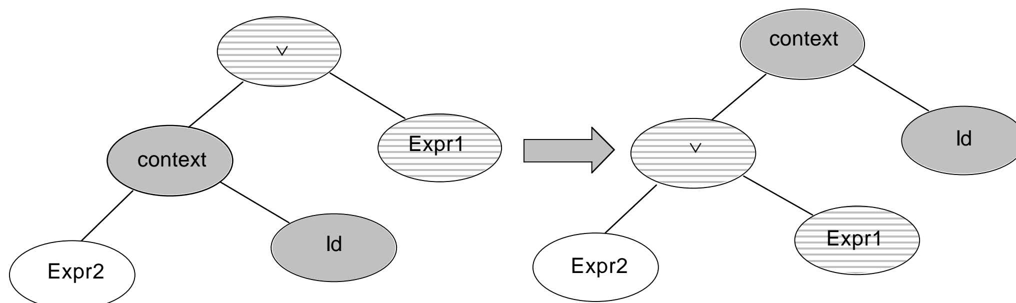

Figure 12: Context Optimization for Disjunction

8.4 Further Optimizations

Further optimizations can be realized by similar transformations:

- Query-Optimizations: As in the case of the optimization of SQL statements, ODL expressions can be optimized by move relations etc. out of the context of quantors. Thus, those basic propositions act like a filter reducing the runtime complexity.
- Functional Optimization: If the assignment of a variable is functional dependent on other assignments of variables, this assignment can – given a constructive definition of this dependency – by calculated instead of found by search.

[4] The optimization is similar to a skolemization of the expression since after optimization the context quantor is dependent on the assignment of universally qunatified variables but not existentially quantified variables.

9 Appendix

9.1 Grammar of CCL-Extension ODL (Operation Definition Language)

Remarks:

- Terminal symbols are **bold-faced**
- Terminals grouped together by lexical analysis are *italic* (e.g. *ident*, *bool_constant*, etc.)
- All other symbols are non-terminals
- Grammar is partially prepared for extensions (see, e.g., definition of type)
- Grammar is defined with respect to readability; for parsabiliy, different production rules may be favorable
- Here, we use complex entities with associated attributes; this is reflected by the selector_expression production

proposition ::= proposition **and** proposition |
proposition **or** proposition |
proposition **implies** proposition |
proposition **equiv** proposition |
neg proposition |
(proposition **)** |
basic_proposition |
quantor_proposition

basic_proposition ::= relation |
equal_expression |
bool_constant

relation ::= pre_relation |
post_relation

pre_relation ::= relation_ident **(** arguments **)**

relation_ident ::= *ident*

post_relation ::= **result** relation_ident **(** arguments **)** |
result not relation_ident **(** arguments **)**

arguments ::= argumentlist |
<empty>

argumentlist ::= argument **,** argumentlist |
argument

```
argument               ::=   expression

expression             ::=   functional_expression |
                             selector_expression |
                             primitive_expression

functional_expression  ::=   function_ident ( arguments )

function_ident         ::=   ident

selector_expression    ::=   variable . selection

selection              ::=   selector . selection |
                             selector

selector               ::=   ident

primitive_expression   ::=   constant |
                             variable

constant               ::=   bool_constant |
                             int_constant |
                             string_constant

variable               ::=   ident

equal_expression       ::=   expression = expression

quantor_proposition    ::=   forall variable_definition . proposition |
                             exists variable_definition . proposition |
                             context variable_definition . proposition |
                             new variable_definition . proposition

variable_definition    ::=   variable : type

type                   ::=   basic_type

basic_type             ::=   type_ident
```

9.2 Extension: Sets

For the definition of more complex operations higher order type constructions like sets are needed. As a result, the binding of variables introduced by quantors becomes more complex. This suggests to introduce additional classes extending those introduced in section 4. Generally, there will be no distinction between sets and types, sets will be treated as a special form of types. Basic types like integer or string are interpreted as the sets of all integers or strings. Based on these basic types, types can be constructed using the type constructors

- Set comprehension: The selection of subtypes by filtering upon types. Example: x: port **with** port.type = inport
- Product: The construction of tuples of types. Example: z: **(** x: integer, y: integer)
- Set types: The construction of typed sets. Example: s: **set** integer

To support these type constructions by the evaluation algorithm, node classes for the nodes of the evaluation tree must be introduced. The super class Type is introduced to support the binding of variables of a type by quantors, context quantor or new operator. Furthermore, types can be used to implement an equality relations.

- Class Type: Abstract class used to access all values of a given type (set of entities), is implemented by concrete classes
  - Methods: context: to query user for input as needed by the context quantor; next: used to generate all instances of the class one after the other to be used in an existential or universal quantor.

All possible types in the ODL language are subclasses of this abstract class. All basic types are either model entities or basic attribute types like strings or integers:

- Subclass integer: context: queries for an integer; next: undefined/exception
- Subclass string: context: querys for a string; next: undefined/exception
- Subclass ModelEntity: this class supplies the interface to the product model context: queries for an existing entity as defined by the meta model to be specified by the user; next: if reevaluated lists all existing entities.

As an extension, compound classes are introduced:

- Class Comp: Class to generate a set comprehension given a proposition. query and next methods use the query and next methods of the comprehended type, repeatedly if element does not met proposition.
- Class Product: context: queries for all elements of the product, using their query methods. next: using the next methods iteratively to generate all elements.
- Class Set: Class to generate all elements of the power set. Uses the context and next methods of the class of the base type repreatedly to generate a set instance.

For the extended type system, the grammar given in section 9.1 must be modified as follows:

type	::=	fixed_point_type | set_type | product_type
fixed_point_type	::=	**fix** *ident* set_type
set_type	::=	**set** product_type **with** CCL_proposition
product_type	::=	basic_type | (basic_type_list)
basic_type_list	::=	ident **:** basic_type | basic_type_list **,** ident **:** basic_type

basic_type ::= *type_ident*

Here, CCL_proposition corresponds to proposition as defined in section with the clauses containing **result**, **not result** and **new** removed. Furthermore, a fixed point constructor was introduced to support an equivalent of a recursive definition without introducing the possibility of non termination in the operational interpretation of the language.[5]

9.3 The ODL Model – An Example

For a better understanding of the AQuA-language, we look at the following example:

exists comp:Component. isName(comp,“Master“)

For the treatment of this ODL-term, we have to

- build an ODLTerm object for the quantor node “exists” with the variable “comp”
- build an interface ODLModelEntitiy for the access to the model via the ODLType associated with “Component”
- build an ODLModelRelation for the basic model relation associated with “isName”
- possibly build an ODLVariable object for “comp” (depending on implementation)
- possibly build an ODLConstant object for „Master“ (depending on implementation)

and set the according references to build an appropriate evaluation structure.

Here, two classes as defined in the MetaModel are used:

- Component: the MetaModelElement representing a component of the model
- hasName: the basic relation associated with the MetaModelAttribute „Name“ of the MetaModelElement Component.

To compile the ODL-expression, we have to allocate the appropriate MetaModelElement and MetaModelAttribute, resp. For each MetaModelElement the ODL runtime system of AQuA offers an appropriate interface class ODLModelEntitiy; if an appropriate interface class is found, an association is created. Otherwise a compile error is generated. A similar mechanism is used for MetaModelAttributes to support a ODLModelRelation class for the has<MetaModelAttribute>-relation. Accordingly, appropriate classes have to be found for associatitions to be treated as basic relations.

The structure will be evaluated by calling the evaluate-method for the root object of class ODLQuantor. The argument of the method is the empty variable binding. The evaluation of this object will result in a loop. In each loop an evaluation of the

[5] The evaluation of fixed-point constructs by use of the Kleene chain and delta sets natiurally leads to simple terminating evaluation srategy for such constructs. On the other hand, recursive predicates do not have such a naturally induced strategy to avoid termination but could be extended by cutting of recursive calls with the same arguments. Since both proofs of termination rely on the fact that recursion can only occur over a finite universe, both strategies are essentially alike.

associated ODLTerm object representing the quantified term takes place by calling its evaluate-method with the new binding object. This new binding object is constructed by adding a binding for „comp“ to it with a reference to a different Component object for each loop. To define those references, the query-method of the ODLModelEntitiy associated with Component is used. The ODLTerm of the quantified term here is an ODLModelRelation which is derived as described above. For its evaluation the corresponding ODLModelRelation object evaluates its arguments. The evaluation of the variable comp returns the Component-object associated with it in the current binding; the evaluation of the constant returns its value „Master“. The ODLModelRelation then evaluates whether comp.Name equals „Master“ and returns this value as the Boolean result. Additionally, the witness set is updated depending on the result.

The following picture shows an evaluation tree for the example with links to the model (grayed-out dashed boxes). Additionally, the associations used for the evaluation are marked („Evaluation(...)“).

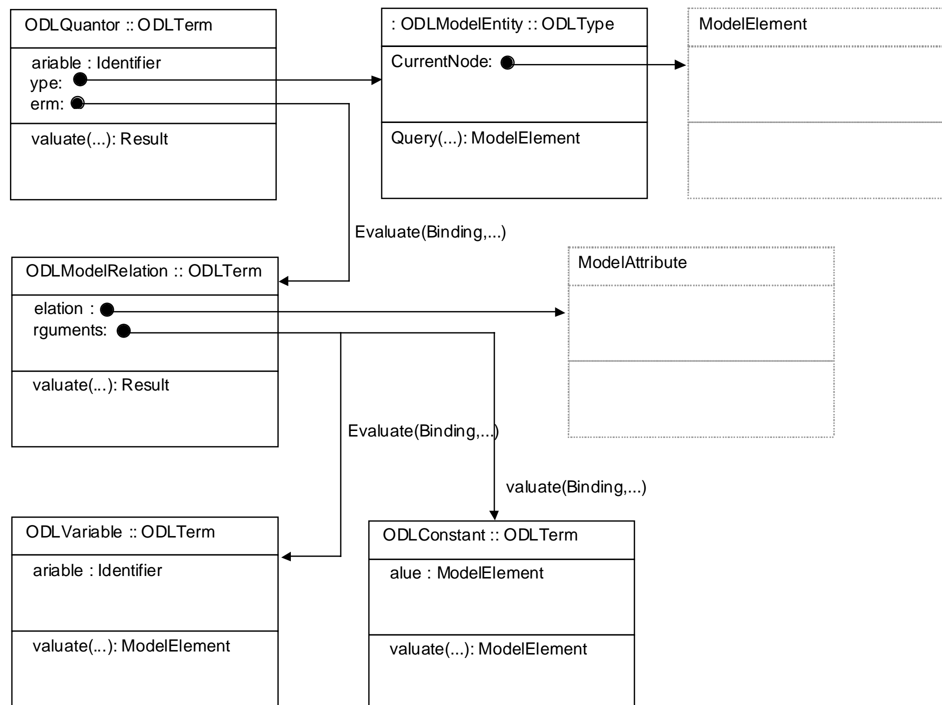

Figure 13: Example of a ODL-Evaluation Tree Structure

Figure 14 illustrates the connections between the ODL-definition of an operation (using the ODL syntax), the ODL model generated during compile-time and used during the run-time evaluation, the meta model used to construct the ODL model, and the model evaluated and modified by the evaluation of the ODL model.

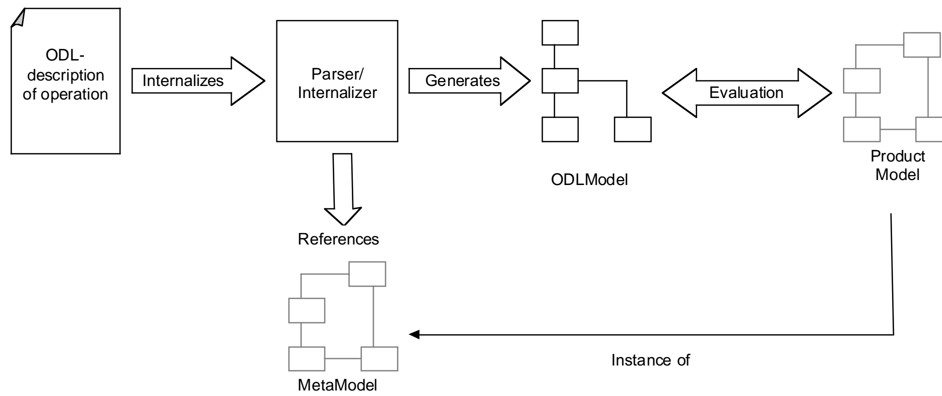

Figure 14: ODL Model, Meta Model and Product Model

9.4 The AutoFocus/Quest Meta-Model

Figure 15 shows a simplified version of an actual AutoFocus/Quest meta model as used in the AQuA framework. For a description of the meta model, see [BL+00].

Figure 15: The AutoFocus/Quest Meta Model

10 Bibliography

[BL+00] Braun, P. Lötzbeyer, H. Schätz, B. Slotosch, S. *Consistent Integration of Formal Methods*. In: Tool and Algorithms for the Construction and Analysis of Systems (TACAS 2000). Susanne Graf, Michael Schwartzbach (eds.). Springer Verlag, 2000.

[FS97] Fowler, M. Scott, K. *UML Distilled*. Addison-Wesley, 1997.

[HS99] Huber, F. Schätz, B. *Integrating Formal Description Techniques*. IN: FM'99 -- Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, Volume II. Jeannette M. Wing, Jim Woodcock, Jim Davies (eds.). Springer Verlag, 1999.

[HS01] Huber, F. Schätz, B. *Integrated Development of Embedded Systems with AutoFocus*. Technical Report TUMI-0701.. Technische Universität München, Fakultät für Informatik, 2001.

[HSE97] Huber, F. Schätz, B. Einert, G. Consistent Graphical Specification of Distributed Systems. In: FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313, pp. 122 - 141, John Fitzgerald, Cliff B. Jones, Peter Lucas (ed.), Springer., 1997.

[Sch00] Scherer, M. A Formalization of OCL with Isabelle/HOL. Diploma Thesis, Technische Universität München, 2000.

[SGW94] Selic, B. Gullekson, G. Ward, P. *Real-Time Object Oriented Modelling*. Wiley, 1994.