A Comparison of Textual Modeling Languages: OCL, Alloy, FOML

Mira Balaban[1], Phillipa Bennett[2], Khanh Hoang Doan[3], Geri Georg[2], Martin Gogolla[3], Igal Khitron[1], Michael Kifer[4]

[1] Computer Science Department, Ben-Gurion University of the Negev, ISRAEL
mira@cs.bgu.ac.il, khitron@cs.bgu.ac.il
[2] Computer Science Department, Colorado State University, Fort Collins, CO, USA
georg@CS.ColoState.EDU, Phillipa.Bennett@colostate.edu
[3] Department for Mathematics and Computer Science, University of Bremen, Germany
gogolla@informatik.uni-bremen.de, doankh@informatik.uni-bremen.de
[4] Department of Computer Science, Stony Brook University, NY, USA
kifer@cs.stonybrook.edu

Abstract. Textual modeling languages are used in model-driven engineering for a variety of purposes. Among the most important purposes are querying a model and formulating restrictions like state invariants or operation pre- and postconditions. This paper compares three such languages. OCL augments UML as a precise language that provides constraint and object query expressions that cannot otherwise be expressed by a diagrammatic notation. Alloy is a simple but expressive logic based on the notion of relations. FOML is a logic rule language that supports object modeling, analysis, and inference. The paper shows typical models in each of the three languages and discusses similarities of and differences between the languages.

1 Introduction

Textual modeling languages are used in model-driven engineering for a variety of purposes, among the prominent ones, querying a model and formulating restrictions like state invariants or operation pre- and postconditions. This paper provides a comparison between three such languages. OCL is augmenting UML as a precise language that provides constraint and object query expressions that cannot otherwise be expressed by diagrams. Alloy is a simple but expressive logic based on the notion of relations. FOML is a logic rule language that supports object modeling, analysis, and inference. The paper shows typical models in each of the languages and discusses similarities and differences among the languages.

This paper is organized as follows. Section 2 introduces the general comparison criteria used in this paper. In sections 3, 4, 5 the three languages are described, using three typical models. In Section 6 the three languages are compared, and section 7 concludes the paper.

2 Modeling Criteria

Modeling tools use *modeling languages* for expressing software models and to provide means for model management and for solving modeling questions. Accordingly, our comparison of modeling languages has two aspects: (1) mode of usage and problems being solved and (2) the representation aspects.

The mode of usage includes constraining a model, querying and analysis, checking satisfiability of constraints, instance creation or completion, instance validation (testing), multiple domains and levels of modeling, and others.

To compare the representations, we consider five categories: *navigation through the elements of the models, support for collections, aggregation, recursion, and subtyping/instantiation. Navigation* refers to traversal of inter-references among elements of a model. Many languages involve special kinds of *navigation expressions* that enable direct reference among objects. These navigation expressions can vary in the amount of flexibility and control, such as support for filtering of undesirable references or use of wild-cards over navigation paths.

The *collections* and *aggregation* criteria refer to the ability of a language to express known kinds of collections, specify new kinds of structures, and aggregate answers to queries using such collections. Support for *recursion* means the ability to define recursive modes of computation. *Subtyping* (or *type hierarchy*) refers to specification of inclusion relationships among types, and *instantiation* refers to specification of instances of abstraction, like application of predicates or functions to data, or specification of class instances.

Sections 3, 4, and 5 introduce each of the three surveyed languages using typical models, keeping in mind the above representation criteria and the typical modes of usage. Section 6 then compares these languages.

3 Modeling with OCL

3.1 OCL Concepts

The Object Constrains Language (OCL) [1, 2] is a textual, descriptive expression language. OCL is side effect free and is mainly used for phrasing constraints and queries in object-oriented models. Most OCL expressions rely on a class model which can be expressed in a (graphical) modeling language like UML [3], MOF or EMF. The central concepts in OCL are objects, object navigation, collections, collection operations and boolean-valued expressions, i.e., formulas.

Objects: An OCL expression will often begin with an object literal or an object variable and denotes a value, an object or a collection of these entities. For example in the context of classes Researcher and Paper and an association submission(author:Researcher, submission:Paper) that also defines rolenames, one could use the objects ada, bob, cyd of type Researcher and subICSE, subMODELS of type Paper. Furthermore variables like p:Paper and r:Researcher could be employed.

Object Navigation: Object navigation is realized by using role names from associations or object-valued attributes which are applied to objects or object collections. For instance, the following navigation expressions could be stated as `ada.submission` or `subICSE.author`.

Collections: Collections can be employed in OCL to merge different elements into a single structure containing the elements. There are four collection kinds: sets, bags, sequences and ordered sets. Sets and ordered sets can contain an elements at most once, whereas bags and sequences may contain an element more than once. In sets and bags the element order is insignificant, whereas sequences and ordered sets are sensitive to the element order. For a given class, the operation allInstances yields the set of current objects in the class.

Collection Operations: There is a number of collection operations which contribute essentially to the expressibility of OCL and which are applied with the arrow operator. Among further operations, collections can be tested on emptiness (isEmpty, notEmpty), the number of elements can be determined (size), the elements can be filtered (select, reject), elements can be mapped to a different item (collect) or can be sorted (sortedBy), set-theoretic operations may be employed (union, intersection), and collections can be converted into other collection kinds (asSet, asBag, asSequence, asOrderdSet).

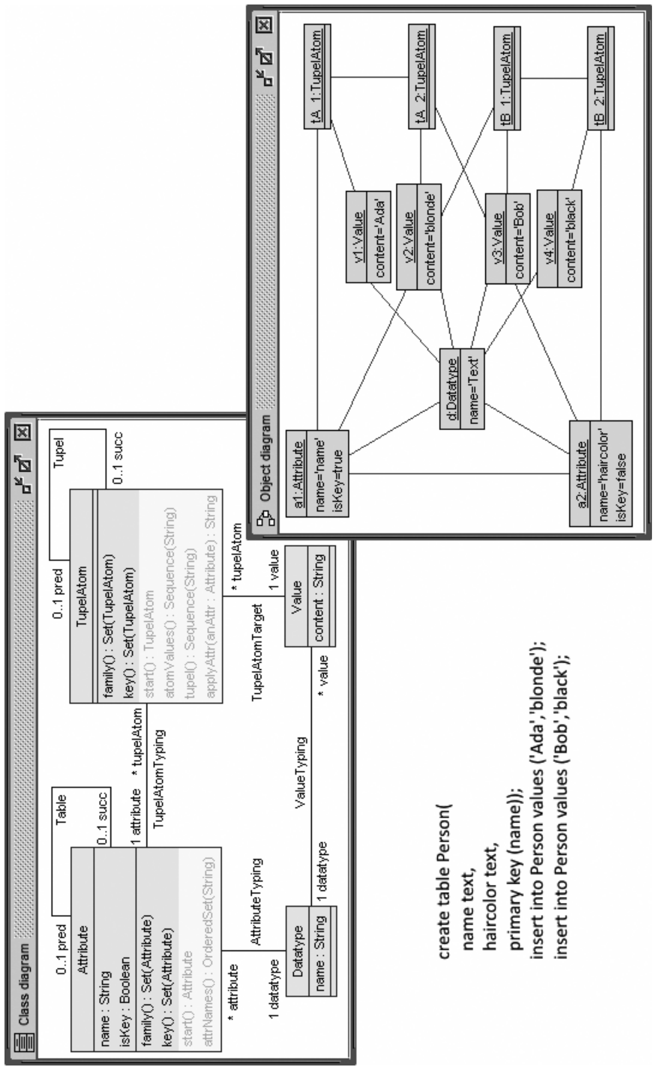
Boolean-Valued Expressions: Because OCL is a constraint language, boolean expressions which formalize model properties play a central role. Apart from typical boolean connectives (and, or, not, =, implies, xor), universal and existential quantification are available (forAll, exists). Boolean expressions are frequently used to describe class invariants and operation pre-and postconditions. In postcondition expressions, the suffix @pre serves to access attribute and role values at precondition time.

3.2 OCL Example Model: The Relational Data Model

This UML and OCL model describes the schema and state aspect of the relational data model. It allows to define relational schemas incorporating tables, attributes and data types. The model also pictures the state aspect in the sense that tuples, tuple constituents (called tuple atoms below) and data type values are characterized. One remark concerning spelling: Because the English word 'tuple' is a reserved word in OCL, the model and this text will employ the German spelling 'tupel' whenever 'tuple' occurs in names of entities that are part of the model (association names, role names, attribute names).

The class diagram for this model is displayed in Fig. 1. The light gray shaded parts show operations that are added for convenient query formulation only and that are not needed in the constraints. The figure also shows a valid example object diagram that would result from the SQL statements indicated in the lower left part.

Observe that three OCL collection kinds, `Set`, `OrderedSet`, `Sequence`, occur in the operations and are thus conceptually needed. The fourth, `Bag`, would occur if, for instance, the order in the operation `tupel():Sequence(String)` is not relevant in a respective expressions or query. One could then apply additionally the conversion operation `asBag()` resulting in `tupel():Bag(String).` Thus all



```
create table Person(
    name text,
    haircolor text,
    primary key (name));
insert into Person values ('Ada','blonde');
insert into Person values ('Bob','black');
```

Fig. 1: Class diagram and object diagram.

4 OCL collection types occur in the example and must be employed in order to deliver conceptually different result.

3.3 Constraints of the Relational Data Model

A constraint in OCL must be formulated at the class level and in the context of a particular class (specified by the keyword *context*). There are three types of constraints: *invariant*, *postcondition* and *precondition*. An invariant is a constraint that states a condition that must always be true; it is recognized by keyword *inv*. A precondition must be true just prior to the execution of an operation and a postcondition must be true just after the execution of an operation. The keywords *pre* and *post* are used to formulate preconditions and postconditions, respectively. A number of constraints have been formulated for the relational data model. To give examples, we introduce several of them in this section. The first constraint is a technical requirement that arises due to modeling with reflexive associations. The constraint guarantees that the order of `Attribute` objects defining a `Table` object is acyclic.

```
context a:Attribute inv acyclicTableLinking:
  Set{a.pred}->closure(pred)->excludes(a) and
  Set{a.succ}->closure(succ)->excludes(a)
```

In this expression, we use the transitive closure operation to go through `Attribute` instances through the `Table` link. The second constraint ensures that different `Attribute` objects have different names within a `Table`.

```
context a:Attribute inv uniqueAttributeNamesWithinTable:
  a.family()->forAll(a1,a2 | a1<>a2 implies a1.name<>a2.name)
```

Another constraint for the model is to guarantee the set of key `Attribute` objects within a `Table` is not empty.

```
context a:Attribute inv keyAttributesNotEmpty:
  a.family().key()->notEmpty
```

We also formulate a constraint to specify that two different `Tupel` objects of a `Table` can be distinguished by a key `Attribute` of the `Table`. This constraint is the core requirement of the relational data model expressing that two tuples (or in other words rows) in a table show different values for the key attributes.

```
context t1,t2:TupelAtom inv keyAttributesHaveUniqueValues:
    t1<>t2 and t1.attribute.family()=t2.attribute.family() and
    t1.pred=null and t2.pred=null implies
    t1.attribute.key()->exists(ka |
      t1.applyAttr(ka)<>t2.applyAttr(ka))
```

A number of central OCL collection operations are employed in this model (shown in the order of appearance): closure, excludes, forAll, size, notEmpty, exists.

Verifying and validating UML and OCL models like the above one is supported by the design tool USE (UML-based Specification Environment) [4, 5].

4 Modeling Examples: Alloy

4.1 Brief Description of Alloy

The declarative Alloy language [6], [7] can be used to specify the structure of a system textually, and the Alloy Analyzer can then be used to explore it. Behavior can be explored using predicates that query the system in various ways. The Alloy Analyzer searches for system instances or counterexamples that satisfy the predicates, assertions, and system constraints that have been specified.

The central concepts of Alloy are called **signatures** (their instances are called **atoms**) and the relations between them. We have created a RolePermissionEmployee model that demonstrates navigation and transitive closure, in addition to constraints and predicates to explore the model. We define a signature *Sys*, to specify the relations among the other signatures, and facts about the sets of signatures, their relations, and other constraints to which the system must comply. The Alloy Analyzer uses a SAT solver or the Kodkod model finder to find instances or counterexamples of predicates and assertions. The search space is bound through a scope that limits the number of elements of each signature in the model. Experience has shown that a small scope is often able to find the same issues as a larger scope, so limiting the scope to small numbers to make the analysis more tractable is a common approach when using the Alloy Analyzer.

4.2 Modeling Criteria with Alloy

Navigation in Alloy occurs via relations, using the dot operator (which also serves as a relational join operator). Collection and aggregation are specified as sets referenced through the dot operator with optional predicate logic to constrain the results. Alloy handles recursion by unrolling and is limited to a maximum depth of three.

4.3 RolePermissionEmployee model

The RolePermissionEmployee Alloy model, called a **module** in Alloy, is textual. Listing 1.1 shows the specification for a portion of the model that does not contain the facts and constraints of the Sys signature or the predicates used to explore it. The full model can be found at [8].

Listing 1.1: Portion of RolePermissionEmployee Alloy textual model

```
module RolePermissionEmployee
open util/graph[Role] as g_r
sig Name {}
sig Role { roleName: Name }
sig Permission {}
sig Employee {}
sig Sys {
  roles: set Role, perms: set Permission, roleHierarchy: roles-> roles,
  rolePermissions: roles some -> some perms }
```

Alloy keywords are bolded in Listing 1.1. The model name (RolePermissionEmployee) follows the **module** keyword, and signature names follow the **sig** keyword. Signatures can be abstract. They can also be restricted to a single instance using the keywords **lone sig**, and hierarchies are supported (**extends** keyword). Relations are specified with the name of the relation (e.g. rolePermissions), and the participants in the relation separated with the $->$ symbol. The keyword **some** specifies that there must be at least one instance of the signature indicated. The keyword **open** allows other file contents to be imported, here, Alloy-supplied graph utilities that define a forest, tree, and acyclic graph; these are used in the model *Sys* signature constraints, where roleHierarchy is defined as a forest (*forest[roleHierarchy]*).

The RolePermissionEmployee model can only be visualized as an instance using the Alloy Analyzer; a visualization of the Alloy model signatures and relations is not possible. However, the similarity between the concepts of signatures and classes allows us to create a representation of the Alloy model as a UML model. Figure 2 shows the complete RolePermissionEmployee model.

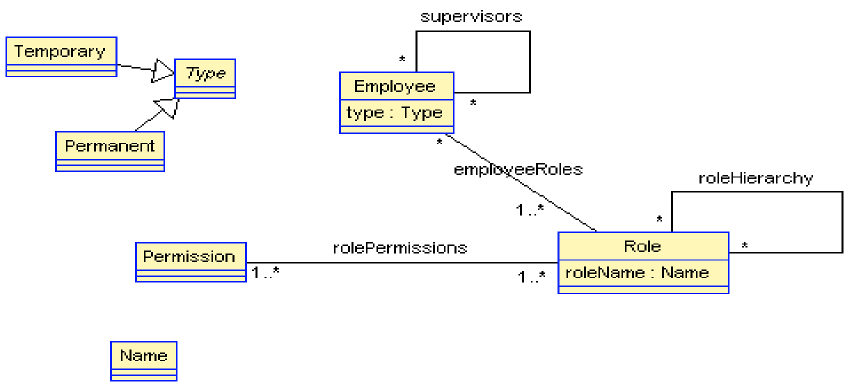


Fig. 2: The RolePermissionEmployee Alloy model as a UML class diagram

4.4 Constraints of the RolePermissionEmployee Model

Two system constraints are shown in Listing 1.2. Brief explanations of the constraints are shown as comments (//) in the listing. Transitive closure is provided by Alloy (ˆ in Listing 1.2), so the constraint that accesses the role hierarchy is straightforward to specify. We can also write predicates to look for instances with particular characteristics, for example, a predicate that an employee has all the roles of its supervisees. This predicate, a predicate it calls, and the Alloy Analyzer command to search for an instance that exhibits this behavior are shown in Listing 1.3.

Listing 1.2: Some Sys signature constraints

```
roleNames = roles.roleName
all n: roleNames | lte[#roleName.n, 1] // role names are unique
all r1, r2 : roles | some r1->r2 & (^roleHierarchy ) implies
  no r1.rolePermissions & r2.rolePermissions // permissions not repeated
```

Listing 1.3: A RolePermissionEmployee model predicate

```
pred twoSigsRelated_WithConstraintRelation
(sig1, sig2: univ, rel: univ->univ) {
let al = allRelations[] | no sig1.rel and //allRelations is a function
some rel.sig2 and no sig1->sig2 & al and some sig1->sig2 & ^al }
pred employeesHasRoleThroughDescendantRole
  (sig1: Employee, sig2: Role) {
  twoSigsRelated_WithConstraintRelation[sig1, sig2, Sys.roleHierarchy] }
run employeesHasRoleThroughDescendantRole expect 1
```

5 Modeling with FOML

FOML [9] is an expressive logic rule language that provides intensional and executable formal basis for software models. It naturally supports model-level activities, such as constraints (extending UML diagrams), dynamic compositional modeling (intensional, transformational), analysis and reasoning about models, model testing, design pattern modeling, specification of Domain Specific Modeling Languages, and meta-modeling. Meta-modeling in FOML relies on uniform treatment of types and instances and spans both definition of abstract syntax and semantics. As an executable modeling language, FOML can express and reason about multiple crosscutting multilevel dimensions, including instantiation constraints.

Technically, FOML is a semantic layer on top of a compact logic rule language of *guarded path expressions*, called *PathLP*, an adaptation of a subset of F-logic [10]. In the overall schema of things, PathLP [11] provides reasoning services over unrestricted instance-of and subtype relations and over typed object-link relations while FOML [12] provides the modeling framework. The PathLP language consists of *membership* and *subtype* expressions, *path expressions* for objects and for types, *rules*, *constraints*, and *queries*, and is implemented on top of the XSB logic reasoning engine.

Below we briefly review the main features of FOML and show how it can be used for modeling, reasoning, and testing. All examples refer to the class model in Figure 3. Details of the syntax and semantics of FOML can be found in [9] and the precise representation of the class model in the figure is found in [13].

Path Expressions: The main syntactic construct in the language is an *object path expressions*. The basic form of a path expression is `root.link[guard]`, where `root, link`, and `guard` are terms that denote semantic entities, and `link`, applied at `root`, evaluates to a set that contains `guard`. For example,

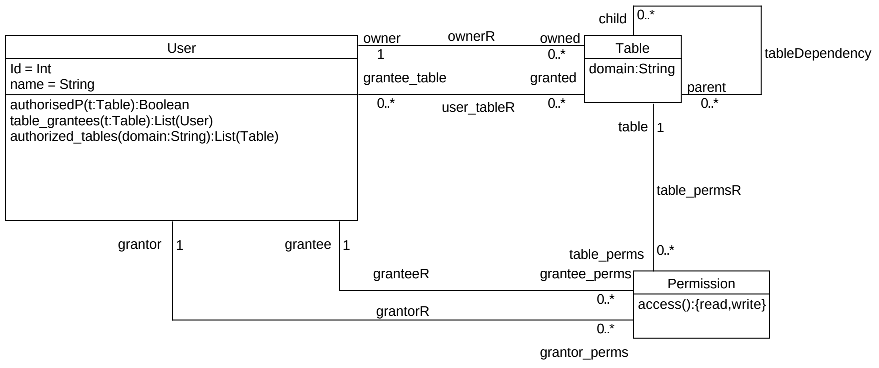


Fig. 3: A User-Table-Permission class model

```
t.owner[?User].grantor_perms[?Per].table[t]
User.property[?p]
```

are (object) path expressions that can be used for: (1) the first expression, given a table object `t`, to retrieve pairs of a user `?User` and a permission `?Per` such that `?User` is an owner of `t` and a grantor of an access permission `?Per` to `t`; (2) the second expression, retrieving properties of class `User`. Symbols that are preceded with "?" are variables, that can be instantiated by terms.

Facts, Rules, Constraints, Queries: FOML uses the regular Logic Programming nomenclature of facts, rules, constraints and queries.

Facts are used to state data. For example, the facts

```
mary.owned[t1];             t1.table_perms[p1];             (1)
p1.grantee[john];           mary.grantor_perms[p1];         (2)
```

state that `mary`'s `owned` table is `t1`, which has permission `p1` in which `mary` is the `grantor` and `john` is the `grantee`. The first three facts can be combined into a single navigation fact:

```
mary.owned[t1].table_perms[p1].grantee[john];
```

Membership and Subtyping: Membership and class hierarchies are modeled using the relations ":" and "::", respectively. For example:

```
t1:Table;      mary:User;                                (3),(4)
t2:Table;      john:User;                                (5),(6)
SystemTable::Table;                                          (7)
Table:Class;   User:Class;   User.prop[owner];               (8)
```

are facts that state that `t1,t2` are `Table`-objects, `Table` itself is a `Class`-object that has a subclass `SystemTable`. The fact (7) is not shown in the diagram and the facts in (8) talk about the meta-level. The facts (3)-(8) describe data in three different layers in the OMG classification: Data, Model, and Meta-model.

Rules represent implications and are denoted by the symbol `:-`, which separates the *head* (conclusion, on the left) from the *body* (premise, on the right). Rule 9, below, states that if a user `?u` is a `grantee` in some permission access to a table `?t`, then `?u` is `granted` access to `?t`. Rule 10 states that a child table has the domain of its parent table.

```
   ?u.granted[?t] :- ?u.grantee_perms.table[?t];                        (9)
   ?t.domain[?d] :- ?t.parent.domain[?d];                              (10)
```

The meta-level relationships between an association and its respective pairs of properties and classes, including multiplicities, are specified by two *path expressions*, one for each property of the association. For example, the association `grantorR` between classes `User` and `Permission`, with properties `grantor` and `grantor_perms` and multiplicities 1..1 and 0..*, respectively, is expressed by:

```
   grantorR.prop(grantor,1,1)[User];
   grantorR.prop(grantor_perms,0,*)[Permission];
```

Queries are recognized by the prefix `?-`. They are used to retrieve information that is implicit in the specification. For example,

Find all grantor-grantee-permission triplets to tables in the teaching domain:

```
   ?- ?u:User, ?u.grantor_perms[?p].grantee[?v],
      ?p.table.domain["teaching"];
```

Find pairs of grantor-grantee `?u, ?v` *to permissions for table ?t*

```
   ?- ?u.compose_via_obj(grantor_perms,?p,grantee)[?v],?p.table[?t];
```

The second query uses a higher-order property constructor `compose_via_obj`. Another higher-order constructor is `closure`. Both are defined below:

Objects ?o and ?v are related by compose(?p1,?p2) if there is a ?p1.?p2 path from ?o to ?v:

```
   ?o.compose(?p1,?p2)[?v] :- ?o.?p1.?p2[?v];
   ?o.compose_via_obj(?p1,?u,?p2)[?v] :- ?o.?p1[?u].?p2[?v];
```

closure(?p) is a property that is the transitive closure of ?p:

```
   ?o.closure(?p)[?v] :- ?o.?p[?v];
   ?o.closure(?p)[?v] :- ?o.?p.closure(?p)[?v];
```

Constraints are recognized by the "`!-`" sign. They specify forbidden states. The following example, involves forbidden grantor-grantee states: *A user* `u` *cannot be granted an access to a table* `t` *from grantor* `v` *that was granted (directly or indirectly) access to* `t` *from* `u`*.* That is, the composition of the `grantor_perms` and `grantee` properties, with respect to `t` is acyclic:

```
?u.grantor_grantee(?t)[?v] :-
    ?u.compose_via_obj(grantor_perms,?p,grantee)[?v],?p.table[?t];
!- ?t:Table, grantor_grantee(?t).circular[true];      (11)
```

where circularity of a property is defined, at the meta-level, by:

```
   ?p.circular[true] :- ?o.closure(?p)[?o];                      (12)
```

Another example of constraints involves the status of class `Permission` (with its associations `granteeR` and `table_permsR`) as an association class with respect to association `user_tableR`.

Every pair of a granted user ?u to a table ?t has a corresponding permission:

```
   !- ?u.granted[?t], not ?u.grantee_perms.table[?t];           (13)
```

For every grantee user ?u to a table, there is a single corresponding permission:

```
   !- ?u.grantee_perms[?p1].table[?u.grantee_perms[?p2].table],
      ?p1!=?p2;                                                  (14)
```

The other direction of the association class constraint, that for every permission, its user and table are related by user_tableR is handled by inference rule (9).

Type path expressions: FOML contains a construct that defines type level implications and constraints. For example, the type path expression

```
   Permission!grantor[User]{1..1};
```

means that a permission must have a single user as a grantor.

Class operations: Rules are used to specify class operations. For example, the operation `table_grantees` on class `User` that computes sequences of grantees to a table `?t` initiated (directly or indirectly) by a user `?u` is defined as follows:

```
?u.table_grantees(?t)[?object_path] :-
    ?u.path(compose_via_obj(grantor_perms,?t,grantee))[?object_path];
```

where the `path` property constructor is defined by these rules:

```
?o.path(?p)[?v] :- ?o.?p[?v];
?o.path(?p)[ [?v|?path] ] :- ?o.?p[?v].path(?p)[?path];
```

The `path` constructor can be further restricted to create only simple paths (the ones with no repeated nodes):

```
    ?o.simple_path(?p)[?v] :- ?o.?p[?v];
    ?o.simple_path(?p)[ [?v|?path] ] :-
           ?o.?p[?v].simple_path(?p)[?path], not ?path._member[?v];
```

The `get_access` operation of class `User`, which retrieves the tables with a specified domain to which a user has access is defined as follows:

```
    ?u.get_access(?d)[?tables] :-
           set(?t, (?u.granted[?t].domain[?d]), ?tables);
```

Here `set` is an aggregate operator, which collects all `?t` that satisfy the condition given by the second argument of `set` and returns the list of all such `?t`'s in the variable `?tables`.

6 Language Comparison

6.1 OCL vs. Alloy and FOML

OCL comparison to Alloy: There are many similarities between the ways constraints are expressed in Alloy and OCL. Alloy navigates using the dot operator through relation names, which can be equivalently expressed through association end names in OCL. However, OCL supports n-ary associations and navigation through them, which cannot be done in Alloy via the navigation methods based on relations. The transitive closure functionality from Alloy (denoted with the symbol ˆ in Listing 1.2) is also provided in OCL, so the constraints related to the hierarchy structure, e.g., the *permissions not repeated in the role hierarchy* constraint in the *RolePermissionEmployee* model, can be expressed in OCL as presented in the listing below (where *next* is the end-role of the *roleHierarchy* relation).

Listing 1.4: Equivalent constraint in OCL

```
context Role inv noRepeatedPermissions:
  Role.allInstances()->forAll(r1,r2|r1.next->closure(next)->includes(r2)
   implies r1.permissions->intersection(r2.permissions)->isEmpty)
```

In Alloy, a predicate is a Boolean expression for which satisfying instances are to be produced. In OCL, one can use additional invariants for this purpose. However, Alloy predicates can be parameterized, so one can use a predicate within other logic statements, e.g., predicate or constraint. This is one major difference between Alloy and OCL, because OCL does not directly support this functionality. For example, to also express the *employeesHasRoleThroughDescendantRole* predicate, one has to duplicate the same OCL code by putting it in two different predicates, while in Alloy only one copy of the code would be used (for instance, in a predicate *twoSigsRelatedWithConstraintRelation*), and that same code will be called by whoever needs it. An alternative solution for introducing a specialized predicate in UML and OCL is to extend the model with singleton System class and place a boolean operation in it.

Another difference is the support for sets and collections. While OCL supports one-dimensional sets and other collections, everything in Alloy is a set. For example a relation is a multi-dimensional set of tuples. Therefore we have to transform the set operator (&) on relations in Listing 1.2, to the single set operator for an equivalent constraint (see Listing 1.4).

OCL comparison to FOML: Most of the language features of FOML are supported in OCL. First of all, both languages navigate through association-end names (role names) using the dot operator and both support composite associations (n-ary associations). Therefore FOML can formulate the constraints, queries, and rules on multi-relation models, like the *User-Table-Permission* model. Additionally, one can use variables in both FOML and OCL. However, the FOML variables are untyped whereas the OCL variables must be typed explicitly. Also similarly to FOML, the closure functionality is provided in OCL. Consequently, we can equivalently express the first constraint in the *User-Table-Permission* model, which refers to the hierarchy relation between *users* through the *grantorR* and the *granteeR* associations. That said, the powerful recursion mechanism of FOML goes well beyond transitive closure.

The main difference between the two modeling languages is the multilevel modeling support. While one cannot access the meta-model in OCL, FOML supports three-layer specification: *data, model,* and *meta-model.* This enables model queries such as "find the classes related to class *User*, and their relevant roles." This is not always possible in OCL. However, a full semantic description of FOML will be probably much more involved than the specification of OCL semantics.

6.2 Alloy vs. FOML and OCL

Alloy comparison to FOML: A major difference between the capability of Alloy and FOML is FOML's inclusion of the composition constructs in the language. Compose navigation cannot always be equivalently defined in Alloy. This is because in Alloy, navigation occurs through relation names whereas in FOML (and OCL) it can occur through relation end roles. In the *User-Table-Permission* model, *compose* navigates from a named association end to another named association end. If a model has more than one association between the same classes, *compose* can differentiate them by the role names passed into the function. In Alloy, it is not possible to differentiate between the multiple relations in the *User-Table-Permission* model. However, since each association end is represented by a set in Alloy, the presence of the desired elements in the domains and ranges of all the relations across the model can be checked.

Another issue in writing equivalent *User-Table-Permission* constraints and queries with Alloy is that there is no metamodel. This means it is not possible, for example, to retrieve all the relations that might exist in the model as part of a compose function. Instead, a function must be defined that collects all the relations, named explicitly, and use that function. It may be possible to define the Alloy relations using the *User-Table-Permission* role names as attributes, but this is probably not a general solution to the issue.

As discussed previously, navigation in Alloy uses the dot operator. Additional predicate logic statements specify atoms that adhere to particular constraints. Similarly to FOML, Alloy provides a *closure* language keyword (see Listing 1.2). The utility functions provided with Alloy contain useful specifications such as *acyclic* and the graph functions used in the *RolePermissionEmployee* model. These allow specification of the acyclic constraint on the tableDependency relation, e.g., *acyclic[tableDependency].*

Alloy comparison to OCL: Alloy can be used to specify a model similar to the RelationalData model. Classes must be specified as signatures, and attributes as relations between signatures. There are no Boolean constants in Alloy, so there can be no attributes that hold them. However a predicate invocation does return a Boolean value that can be used in other logic statements. Methods do not exist directly in Alloy. Instead, predicates must be defined and used in model exploration. Alloy provides certain relation multiplicity keywords, such as **some** and **lone**; if no such keywords are used in a definition explicitly then any number of signature atoms is acceptable. Alloy constraints and predicates use the fundamental concept of sets, so the invariants and constraints of the *RelationalData* model can be specified for the Alloy model. However, due to the analyzer's search space issues, the size of integers that can be practically used in a model is fairly small.

6.3 FOML vs. OCL and Alloy

FOML is an expressive logic language, with simple semantics, intended for three modes of usage: a textual modeling language, a language for ad hoc querying of (and more generally reasoning about) models and data, and as a language for expressing constraints on models.

Textual modeling: FOML enables specification of models, data instances, and includes a special sub-language for meta-modeling, i.e., enables specification of a variety of models. OCL does not account for model or data specification. Alloy provides its own concepts of *signatures, constraints* and *predicates*, and can specify models and data, using these concepts. Alloy client models translate their own concepts into Alloy.

Ad hoc querying: OCL is designed specifically as a language of constraints that extends UML models. It can be used for ad hoc querying nonetheless, *if* all queries of interest are included in the model as methods. If a new query comes up, the model has to be extended. FOML includes a sublanguage for constructing queries (more powerful, in fact, than SQL) and queries need not be specified as methods in advance. Alloy is not designed specifically as a query language. However, like OCL, it contains an evaluator to test ad-hoc queries over instances generated from the Alloy model.

Inference: This is an essential usage mode for FOML. For example, rule number (9) in Section 5 imposes part of the association-class constraint on class *Permission* and association *user_tableR*. This way, rather than rejecting an incomplete instance, the system infers the missing association. In OCL and Alloy this requirement must be formulated as a constraint.

Instance-validation, creation and completion: Specification of models and data is typical for Alloy and for FOML. For OCL, specification of data is not covered by the OCL language. Rather, most UML/OCL tools, e.g., [4, 5], provide means for expressing data (e.g., with object-diagrams), and use some sort of a SAT or a constraint solver for validation. In FOML, instance validation is done by checking whether the constraints are satisfied. Alloy and most UML/OCL tools use the solvers for either finding counter examples or verifying a given instance. Instance completion or creation is not supported by FOML, while it is a central service of Alloy and UML/OCL tools.

Comparison of the representation aspect in FOML vs. OCL and Alloy: *Navigation* expressions are central in all three languages. OCL navigations follow the associations in the given model. Alloy and FOML enable also navigation along virtual relations between elements. OCL enables *intermediate filtering* of navigation paths, using its collection operators (e.g., *select*); in FOML, intermediate filtering is done by intermediate guards in path expressions.

OCL distinguishes between two kinds of navigation – individual objects and collections, while FOML is deliberately restricted to individual navigation. For example, the `uniqueAttributeNamesWithinTable` constraint in Section 3, is expressed in FOML, as follows:

```
!- ?a:Attribute,?a.family[?a1].name[?a.family[?a2].name],?a1!=?a2;
```

The constraint states that attributes in a "family" (a table) cannot have common names. Comparison with the OCL formulation in Section 3 emphasizes the differences between individual-based and set-based navigation. For example, it seems that expressing the association-class constraint in Section 5 (constraints (13), (14) and inference rule (9)) in OCL might be quite challenging.

Recursion is supported in Alloy and in OCL via a `closure` operator that is applied to a property (association-end). FOML supports recursively defined rules, which enable user-defined recursive operations, including recursively defined types like trees, graphs, paths and cycles in graphs, etc. The closure operation is just one of a rich variety of possible such structures. In particular, constraint (11) in Section 5 shows a closure that is applied to a virtual (*intensional*) property, defined using the `compose_via_obj` constructor of properties.

Subtyping and instantiation is supported in all three languages. However, OCL and Alloy are confined for two level structures of a model and its instances. OCl is restricted by the UML meta-modeling structure, and Alloy is restricted by its underlying relational logic. FOML enables an unrestricted structure of subtyping and instantiation. Therefore, it can naturally support multilevel modeling and meta-modeling. For example, the acyclicity constraint `acyclicTableLinking` in Section 3, is expressed in FOML as a meta-level constraint: `!- ?pred.circular[true];` (Circularity of properties is defined in rule (12) in Section 5).

6.4 Comparison Summary

The following tables summarize the comparisons of the three languages, discussed above, based on the criteria proposed in Section 2. The comparison is split by the different aspects of *representation* and *usage*.

Representation:

	Navigation	Recursion	Subtyping	Multilevel
OCL	Individual & Collection; intermediate filtering; follows associations and derived associations	Transitive closure	Yes	No
Alloy	Individual; follows associations and virtual relations	Transitive closure	Yes	No
FOML	Individual; intermediate filtering; follows associations and virtual relations; wildcard navigation	User-defined recursion (includes transitive closure)	Yes	Yes

Usage:

	Textual modeling	Querying	Inference	Validation	Instance creation & completion
OCL	Yes	Yes	via tools	Yes	Yes
Alloy	Yes	Yes	No	Yes	Yes
FOML	Yes	Yes	Yes	via constraints	No

7 Conclusion and Future Research

This paper presents a first effort to compare textual modeling languages on the basis of their mode of usage and representation aspects. We have sketched the major aspects of each language, and emphasized similarities, differences, strengths and weaknesses.

It seems that the mode of use of Alloy and OCL is closely related, as Alloy can be viewed as a backend system that supports OCL modeling. On the other hand, Alloy and FOML have complementing modes of use. In particular, FOML enables to a certain degree inference, and multilevel modeling which seem not to be directly supported by the other languages.

The representation aspects of the languages have similarities and differences that call for a more thorough comparison. In particular, there is a need to compare the expressivity of different forms of navigation.

References

[1] Gogolla, M.: Object Constraint Language. In Liu, L., Öszu, M.T., eds.: Encyclopedia of Database Systems. Springer, Berlin (2009) 1927–1929

[2] Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A Definitive Guide. In Bernardo, M., Cortellessa, V., Pierantonio, A., eds.: Proc. 12th Int. School Formal Methods for the Design of Computer, Communication and Software Systems: Model-Driven Engineering, Springer, Berlin, LNCS 7320 (2012) 58–90

[3] Gogolla, M.: Unified Modeling Language. In Liu, L., Öszu, M.T., eds.: Encyclopedia of Database Systems. Springer, Berlin (2009) 3232–3239

[4] Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming **69** (2007) 27–34

[5] Gogolla, M., Hilken, F.: Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In Oberweis, A., Reussner, R., eds.: Proc. Modellierung (MODELLIERUNG'2016), GI, LNI 254 (2016) 203–218

[6] Jackson, D.: Software Abstractions. The MIT Press, Cambridge, Massachusetts (2012)

[7] Jackson, D.: Alloy . `http://alloy.mit.edu/alloy/index.html` (2012)
[8] Bennett, P.: RolePermissionEmployee Alloy Model . `http://www.cs.colostate.edu/~plarreeb/RolePermissionEmployee.pdf` (2016)
[9] Balaban, M., Kifer, M.: Logic-Based Model-Level Software Development with F-OML. In: MoDELS 2011. (2011)
[10] Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of ACM **42** (1995) 741–843
[11] Khitron, I., Kifer, M., Balaban, M.: PathLP: A Path-Oriented Logic Programming Language. The PathLP Web Site (2011) http://pathlp.sourceforge.net.
[12] Khitron, I., Balaban, M., Kifer, M.: The FOML Site. `https://sourceforge.net/projects/pathlp/files/foml/` (2016)
[13] Khitron, I.: FOML coding for User-Table example (2016) https://sourceforge.net/projects/pathlp/files/user class diagram/.