From UML and OCL to Relational Logic and Back

Mirco Kuhlmann and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen
{mk,gogolla}@informatik.uni-bremen.de

Abstract. Languages like UML and OCL are used to precisely model systems. Complex UML and OCL models therefore represent a crucial part of model-driven development, as they formally specify the main system properties. Consequently, creating complete and correct models is a critical concern. For this purpose, we provide a lightweight model validation method based on efficient SAT solving techniques. In this paper, we present a transformation from UML class diagram and OCL concepts into relational logic. Relational logic in turn represents the source for advanced SAT-based model instance finders like Kodkod. This paper focuses on a natural transformation approach which aims to exploit the features of relational logic as directly as possible through straitening the handling of main UML and OCL features. This approach allows us to explicitly benefit from the efficient handling of relational logic in Kodkod and to interpret found results backwards in terms of UML and OCL.

1 Introduction

Creating complete and correct models is a critical concern. Modeling languages like UML [24] and OCL [30] allow for precisely specifying systems which often result in complex models. The analysis of formulated system properties thus requires tool support. Lightweight model validation approaches allow for agile analysis, since they allow modelers to automatically perform multiple validation tasks at any stage of development. The advantage of lightweight approaches, in contrast to interactive verification approaches, is (a) their applicability, as users do not need be familiar with fields like logical deduction, and (b) their immediateness regarding the feedback. As a consequence, those approaches must be *efficient*.

We analyze properties of UML class models annotated with OCL constraints by analyzing model instances [9], since the existence or non-existence of instances with specific properties allows direct conclusions about the model itself. For efficiently searching model instances, we apply SAT-based techniques [2], i. e., solvers for Boolean satisfiability. This approach requires the connection of UML and OCL with Boolean logic resulting in a bidirectional transformation. However, we make use of an intermediate language, *relational logic*, which is automatically and efficiently handled by the sophisticated model instance finder

Kodkod [28]. Kodkod transforms relational models into SAT formulas and translates solutions fulfilling the SAT formulas back into relational instances.

In this paper, we present the transformation of UML and OCL models into relational models, as well as the backward translation from relational instances into UML model instances. We pursue a natural transformation approach which aims to exploit the features of relational logic as directly as possible through straitening the handling of main UML and OCL features. This approach allows us to explicitly benefit from the efficient handling of relational logic in Kodkod. While explaining the transformation, we focus on important modeling aspects and concepts which have not been concerned or adequately treated in other UML and OCL model validation approaches based on relational logic [1,27], e. g., n-ary associations and association classes at the UML side, as well as the undefined value and essential operations like collect and navigation via n-ary associations and association classes at the OCL side. This transformation approach is supported by a tool classified as a model validator which processes a class diagram and OCL invariants as well as information (in form of partial object diagrams and properties like the minimum and maximum number of objects and links, or attribute value domains) determining the search space, that is, the set of model instances to be examined. The transformation is fully automated with respect to both directions, from UML and OCL to relational logic, and back from relational solutions to UML (object diagrams) (for an overview see [15]).

The rest of the paper is structured as follows: Section 2 introduces relevant concepts of relational logic and Kodkod. The main Sect. 3 presents the bidirectional transformation. In Sect. 3.1 we consider the transformation of UML class diagrams into relational models, while Sect. 3.2 discusses the backward translation. The configuration of search spaces is shortly sketched in Sect. 3.3. Section 3.4 covers the OCL part of the transformation. Related work is discussed in Sect. 4 before we conclude with Sect. 5.

2 Background: Relational Logic and Kodkod

Relational logic [10] is based on flat n-ary relations, i. e., sets of tuples of atomic values (atoms). The evaluation result of a relational formula thus depends on concrete instances of relations. Atoms are constants with no specific semantics or inner structure. The individual meaning of an atom emerges from its occurrence in specific relations. However, it is possible to assign a specific semantics to a subset of the available atoms by mapping them to integer values. Thereby, integer calculations are enabled. Relations can express three kinds of values:

Atomic Values: An atomic value is represented by a unary relation including exactly one tuple with one component holding the respective atom. For example, the integer value 3 and an atom symbolizing the name Ada are realized by the relational values `[[3]]` and `[[Ada]]`.

Sets of Atoms: A set of atomic values yields a unary relation with possibly more than one tuple or no tuple, in the case of an empty set. The set of atoms

{Ada,Bob,Cyd}, for example, results in the relational value `[[Ada],[Bob],[Cyd]]`. The atoms `Ada`, `Bob` and `Cyd` do not have a specific meaning, unless they are put into a context, e. g., if we declare a unary relation `fNames`, we consider all tuples within instances of this relation as individual first names.

Sets of Relationships between Atoms: Atomic values are often semantically related to other atomic values. This fact can be described with n-ary relations in which tuples hold sequences of atoms. Each position in a n-ary tuple has a specific meaning. Consider, for example, persons who have a name and possibly younger siblings. In order to relate a person to a name and her younger siblings, we can declare two binary relations `fName` and `ySiblings`, and determine that the first position of the tuples in both relations yields a person atom and the second position yields a name or another person atom, respectively. Possible instances could be `fName=[[p1,Ada],[p2,Cyd],[p3,Bob], [p4,Dan]]` and `ySiblings=[[p1,p3],[p1,p4]]`.

Relational logic provides: (a) relational operations like the relational join, product and transitive closure, as well as multiplicity predicates like 'some' and 'lone', (b) set comprehension, (c) set operations like union and subset, (d) Boolean operations like conjunction and implication, (e) quantifiers of first order logic-like existential and universal quantifiers, and (f) integer operations like addition and comparison predicates. The relational join (expressed by a dot `.`) is a central operation, since it allows for extracting and merging the information provided by relation instances. A join is performed in the context of two relational values `x` and `y` which may be of different arity. The evaluation result of the expression `x.y` is equal to $\{(x_1,\ldots,x_{n-1},y_2,\ldots,y_m)|(x_1,\ldots,x_n) \in x \wedge (y_1,\ldots,y_m) \in y \wedge x_n = y_1\}$. An example for information extraction with a join is the determination of a person name based on the mentioned relation `fName`. The expression `[[p1]].fName` results in the name related to the person atom `p1`, i. e., in our example `[[Ada]]`. Another example illustrates the merging of two binary relations which in our case results in a set of tuples relating persons to the names of their younger siblings: `ySiblings.fName=[[p1,Bob],[p1,Dan]]`.

Kodkod is a tool which provides an interface to defining relational models and to efficiently finding relational instances fulfilling given relational formulas [28]. A relational model consists of three parts (we will see examples in later sections):

Declarations: A relation declaration determines the name and arity of a relation for which Kodkod searches a valid instance.

Bounds: Kodkod is a *finite* model instance finder, i. e., the *universe* of atoms available for constructing relational values is finite. A relational model includes (a) an a priori, fully determined universe of atoms, and (b) bounds for each declared relation which generally restrict the sets of possible tuples based on available atoms. In this way, a concrete search space is defined.

Constraints: Relational constraints, i. e., formulas, can further restrict the valid instances of the declared relations.

Our approach translates a UML and OCL model into a relational model handled by Kodkod. Results in form of relational model instances presented by Kodkod will be translated back into instances of the UML and OCL model.

3 A Bidirectional Transformation

The aim of translating UML and OCL models into relational models is an efficient search for UML and OCL model instances which fulfill specific user-defined properties. The transformation of UML class diagram and OCL concepts into relational logic is based on three key requirements:

- The transformation of a UML class diagram results in a set of relations. Instances of these relations must structurally allow for representing *all possible* instances of the corresponding UML class diagram.
- Each *valid* instance of the relational model must represent a *valid* instance of the corresponding UML class diagram respecting the given UML and OCL constraints. The same must apply for *invalid* instances.
- The relational model must be formulated as simply as possible, enabling the most efficient processing by the model instance finder (Kodkod).

UML and OCL offer concepts like collections (i. e., sets, bags, ordered sets, sequences, and nested collections) and a three valued logic which are fundamentally different from concepts of relational logic (e. g., solely flat sets and a two-valued logic). For that reason, the first two requirements which concern the completeness and correctness of the translation conflict with the third requirement concerning efficiency. We tackle two different approaches to transforming UML and OCL into relational logic, one giving weight to the first two aspects, the other focussing on the third aspect:

Extrinsic Relational Approach: This approach aims to transform UML and OCL concepts as completely as possible into relational logic enabling, for example, the translation of all kinds of (possibly nested) collections, strings and the associated operations. Furthermore, the three valued-logic of OCL is simulated at the relational level. This virtual *abuse* of relational logic leads to complex relational structures (involving high-arity relations), large search spaces, and hence to losses in efficiency (for details of the extrinsic approach see [14]).

Intrinsic Relational Approach: The intrinsic approach aims to make use of structures directly supported by relational logic, i. e., atomic values, sets of atomic values, and sets of relationships between atomic values (cf. Sect. 2), as well as relational formulas with two truth-values instead of three (as in OCL). On the one hand, this approach naturally results in manageable relational models which can be efficiently processed. On the other hand, it induces several restrictions to the supported UML and OCL features.

In this paper, we present the intrinsic transformation approach and discuss the advantages and disadvantages, practical implications for validation and feasible alternatives. The intrinsic approach has been successfully applied, for example, in the context of role-based access control (RBAC) revealing that the imposed restrictions do not hinder the validation of reasonable models [17].

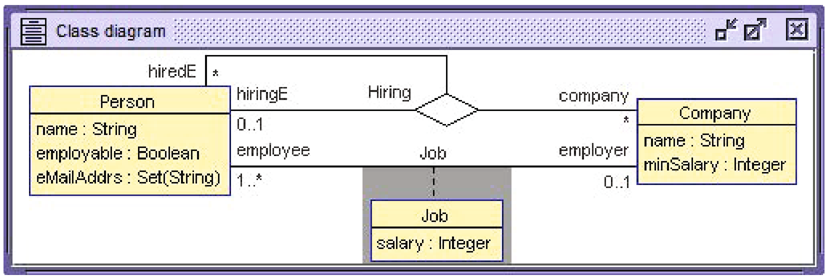

Fig. 1. Example UML Class Diagram

3.1 From UML Class Diagrams to Relational Models

In this section, we focus on the transformation of central UML class diagram features which are frequently used for modeling structural aspects of systems into relational model concepts, i. e., relations and relational constraints. The transformation is illustrated with the help of the example class diagram shown in Fig. 1 which has been designed for explanation purposes covering interesting aspects. It describes persons with a name and a set of email addresses. If employable, persons can have at most one job. A company has a name and defines a minimum salary for its employees. A person can be hired by at most one employee in the context of a specific company. In order to explain both, a binary association and a binary association class, we aim to consider *Job* as an ordinary association (neglecting the grey part), on the one hand, and to consider *Job* as an association class (involving the grey part), on the other hand. The association class adds a salary to each job.

Basic Types. The transformation t uniformly handles the values of the UML basic types *Boolean*, *Integer*, *Real*, and *String* as atomic values. Consequently, basic types result in unary relations whose instances hold the distinctive sets of available basic values, typing the atoms accordingly. Basic type values are needed in the context of UML attribute values, as we will see later.

Boolean $\xrightarrow{t}$ unary relation `Boolean=[[true],[false]]`. The resulting Boolean relation yields a constant instance holding the Boolean values `true` and `false`.

Integer $\xrightarrow{t}$ unary relation `Integer` of structure `[[`i_1`],...,[`$i_{n_{int}}$`]]`.
Example instance: `[[-2],[0],[1],[1000],[1100],[1200],[2000]]`. The integer relation can be variably instantiated, i. e., Kodkod searches an adequate instance. Each integer atom whose name represents an integer literal is bijectively mapped to a corresponding integer value which can be used for calculations within a relational formula. For instance, the atom `1` is mapped to the value 1. Relational logic provides the respective mapping operations (`int` and `Int`). In order to store calculation results in relations, the respective integer values must have an atomic counterpart within the integer relation, e. g., if the result of $1+2$ should be stored as a UML attribute value, the atom `3` must be available.

Real $\xrightarrow{t}$ unary relation `Real` of structure `[[`r_1`],... ,[`$r_{n_{Real}}$`]]`.
Example instance: `[[3.14],[2.71],[1.23]]`. The Real relation is analogously defined to the integer relation, but Real atoms cannot be mapped to processable Real values in relational logic. Thus, Real atoms do not have further meaning except for their comparability, e. g., we can infer that `[[3.14]]` does not equal `[[1.23]]`, but we cannot determine their precedence or apply Real operations.
String $\xrightarrow{t}$ unary relation `String` of structure `[[`s_1`],... ,[`$s_{n_{string}}$`]]`.
Example instance: `[[Ada],[Bob],[Apple],[IBM]]`. Relational logic does not directly support String values with an inner structure, i. e., consisting of sequences of characters. The intrinsic approach handles strings analogously to Real values.
Undef $\xrightarrow{t}$ unary relation `Undef=[[Undef]]`. Primitive values may be undefined. Hence, we need a unary singleton relation holding the undefined value.

Classes and Enumerations. Classes are translated into unary relations with variable instances; enumerations yield unary relations with constant instances:
Class $c \xrightarrow{t}$ unary relation `c` of structure `[[`obj_1`],...,[`obj_{n_c}`]]`, where an atom obj_i (with $1 \leq i \leq n_c$) represents an object identifier.
Example translation: Class *Person* $\xrightarrow{t}$ `Person`.
Example instance: `[[ada],[bob],[cyd]]`.
Enum e=$\{lit_1,\ldots,lit_{n_e}\}$ $\xrightarrow{t}$ unary relation `e=[[`lit_1`],...,[`lit_{n_e}`]]`.
Example translation: Enum *Colors*=$\{r,g,b\}$ $\xrightarrow{t}$ `Colors=[[r],[g],[b]]`.

Associations and Association Classes. The intrinsic transformation fully supports n-ary associations and association classes with multiplicities. An n-ary association has n association ends, where association end i (with $1 \leq i \leq n$) is of type class c_i, i. e., a navigation to this end results in objects of c_i. For translating associations into relational logic we determine a specific order of the association ends in such a way that end i is mapped to tuple position i. Hence, we obtain the following transformation for n-ary associations:
n**-ary Association** $a \xrightarrow{t}$ n-ary relation `a` of structure `[[`obj_{11}`,...,`obj_{1n}`],..., [`obj_{m1}`,...,`obj_{mn}`]]`, where obj_{ij} describes the object occurring in the ith link at the jth association end, plus typing and multiplicity constraints.
Example translation: Association *Hiring* with association end order: *hiringE*, *hiredE*, *company* $\xrightarrow{t}$ `Hiring` plus constraints shown below.
Example instance: `[[ada,bob,apple]]`. n-ary associations result in n typing constraints requiring each association end, i. e., each tuple position, to hold objects of the related class, i. e., atoms of the respective class relation. The universe relation `univ` provided by relational logic including all existing atoms allows us to navigate to the desired tuple positions by cutting off the unneeded tuple positions. Consider the following typing constraints for association relation `Hiring`:

`(Hiring.univ).univ in Person`	the hiring employee (first position) is a person
`(univ.Hiring).univ in Person`	the hired employee (second position) is a person
`univ.(univ.Hiring) in Company`	a person is hired for a company (third position)

Furthermore, each association end yielding a constraining multiplicity differing from *0..** results in a multiplicity constraint. Consider for example the constraint for association end *hiringE* which demands that each pair of objects belonging to the opposite association ends *hiredE* and *company* is connected to at most one object of association end *hiringE*:

```
all c2:Person, c3:Company | #((Hiring.c3).c2)<=1
```

If the lower bound of a multiplicity is greater than 0, the constraint is extended accordingly. Generally we see that the absence of a link is indicated by the absence of a corresponding tuple in the association relation. In this way, the navigation to an association end directly results in set values. Objects not linked to another object do not occur in the set. If no object is connected, the navigation results in an empty set. Binary associations are an exception to this rule if an association end is single-valued, i. e., if it yields the multiplicity *1* or *0..1*. In this case, a navigation to this end results in exactly one object. Multiplicity *0..1* allows this object to be undefined. Thus, the absence of a link is expressed by tuples having the `Undef` atom at the respective position. That is, in contrast to general association relations the absence of a link is not indicated by the absence of the respective tuple, but by the explicit occurrence of the undefined value:

Binary association $a \xrightarrow{t}$ binary relation `a` of structure `[[`obj_{11}`,`obj_{12}`]`,..., `[`obj_{m1}`,`obj_{m2}`]]`, where obj_{ij} may be undefined, if association end j yields multiplicity *0..1*, plus special relational constraints for typing and multiplicities.
Example translation: Association *Job* with association end order: *employee*, *employer* (dismissing the grey association class part) $\xrightarrow{t}$ `Job`.
Example instance: `[[ada,ibm],[bob,ibm],[cyd,Undef]]`. Constraints:

`Job.univ in Person`	the employee is a person
`univ.Job in Company+Undef`	the employer is a defined company or undefined
`all c1:Person|#(c1.Job)=1`	a person is connected to one atom via relation `Job`
`all c2:Company|#(Job.c2)>=1`	a company is connected to at least one person

If we respect the grey part in Fig. 1, we obtain an association class. Association classes yield two relations. One relation represents the class perspective following the same translation rules as relations for ordinary classes. In every respect, the class relations of association classes can be handled like class relations of ordinary classes. The relation representing the association part is translated analogously to ordinary associations, except for an additional column at the first tuple position holding the participating association class objects:

***n*-ary Association class** $ac \xrightarrow{t}$ unary relation `ac` of structure `[[`ac_obj_1`]`,..., `[`ac_obj_m`]]`, n+1-ary relation `ac_assoc` of structure `[[`ac_obj_1`,`obj_{11}`,`...`,`obj_{1n}`]` ,...,`[`ac_obj_m`,`obj_{m1}`,`...`,`obj_{mn}`]]`, plus typing and multiplicity constraints.
Example translation: Association class *Job* with association end order: *job* (implicit), *employee*, *employer* (respecting the grey part) $\xrightarrow{t}$ `Job`, `Job_assoc`.
Example instance of `Job`: `[[job1],[job2]]`. Example instance of `Job_assoc`: `[[job1,ada,ibm],[job2,bob,ibm],[Undef,cyd,Undef]]`. As ordinary association ends, association class ends are typed:

```
(Job_assoc.univ).univ in Job+Undef
```

Furthermore, the association class end requires two multiplicity constraints for ensuring that (a) each permutation of objects corresponding to the opposite ends is connected to at most one association class object, and (b) each association class object is connected to exactly one permutation of *defined* objects:

(a) `all c2:Person, c3:Company|#((Job_assoc.c3).c2)<=1`

(b) `all c1:Job | #(c1.Job_assoc)=1 && (c1.Job_assoc) in (Person->Company)`

Analogously to binary associations, binary association classes need a special handling if single-valued association ends are involved. In the case of an object-valued association end like *employer*, the opposite end (i. e., *employee* in our example) is always related to one object which may be undefined:

(c) `all c2:Person | #(c2.(univ.Job_assoc))=1`

In the case of set-valued association ends like *employee* with multiplicity *1..**, the opposite end (*employer*) is never linked to an undefined association class object because, in this case, the navigation to the association class end results in a set of objects (i. e., one or more jobs in our example):

(d) `all c3:Company|!(Undef in ((Job_assoc.c3).univ)) && #(Job_assoc.c3)>=1`

Attributes. Independent from their types, UML attributes are always translated into binary relations. Attribute relations relate objects with attribute values. If an attribute is not defined, the respective objects are related to the undefined value. In the case of set-valued attributes, we use the special atom `Undef_Set` to indicate the absence of a defined set. This way, we can distinguish between undefined set values (object related to `Undef_Set`), defined set values including the undefined value (object related to `Undef`) and an empty sets (the corresponding object does not participate in the attribute relation instance). Regarding this detail, the translation of attributes and binary associations differ.

Attribute *Class::attr* $\xrightarrow{t}$ binary relation `Class_attr` of structure `[[`obj_1`,` val_{11}`]`,..., `[`obj_1`,`val_{1n_1}`]`,...,`[`obj_m`,`val_{m1}`]`,..., `[`obj_m`,`val_{mn_m}`]]`, where n_i is the number of atoms representing the attribute value related to obj_i ($1 \leq i \leq m$), plus typing and multiplicity constraints. Basic, object and enumeration type attributes require $n_i = 1$ for all i. Set type attributes allow any positive value including 0 for n_i, also n_i and n_j ($1 \leq j \leq m$ and $i \neq j$) may differ.
Example translation: Attribute *Person::name*, *Person::eMailAddrs*, *Job::salary* $\xrightarrow{t}$ `Person_name`, `Person_eMailAddrs`, `Job_salary`.
Example instance (`Person_name`): `[[ada,Ada],[bob,Bob],[cyd,Undef]]`.
Example instance (`Person_eMailAddrs`):
`[[ada,ada@apple.com],[ada,ada@gmail.com],[cyd,Undef_Set]]`.
Example instance (`Job_salary`): `[[job1,2000],[job2,1200]]`.

Attribute relations are constrained by formulas for determining the attribute domain, type and multiplicity. The attribute domain is always a class relation. The undefined value is not involved at the domain side. However, the undefined value always participates in the attributes type definition. Let us consider the constraints for the basic type attribute relation `Person_name`:

```
Person_name.univ in Person          the domain is Person
univ.Person_name in String+Undef    the type is String including Undef
all c:Person | #(c.Person_name)=1   the attribute relates a person to one atom
```

Set-valued attributes yield different constraints:

```
Person_eMailAddrs.univ in Person       the domain is Person
univ.Person_eMailAddrs in              the type is a set of String values in-
  String+Undef+Undef_Set               cluding undefined values
all c:Person |                         an undefined set is not accompanied
  Undef_Set in c.Person_eMailAddrs =>  by other values
    #(c.Person_eMailAddrs)=1
```

3.2 From Relational Instances to Class Diagram Instances

In this section, we consider the straightforward backward translation of a valid relational model instance provided by Kodkod into instances of UML class diagram concepts. We illustrate the transformation with the help of instances of relations resulting from the example class diagram shown in Fig. 1 including the grey association class part:

```
Boolean=[[true],[false]],            Integer=[[1000],[1100],[1200],[2000]],
String=[[Ada],[Bob],[Apple],[IBM]], Undef=[[Undef]],
Person=[[ada],[bob],[cyd]],          Company=[[apple],[ibm]],
Job=[[job1],[job2]],                 Hiring=[[ada,bob,apple]],
Job_assoc=[[job1,ada,apple],[job2,bob,apple],[Undef,cyd,Undef]],
Person_name=[[ada,Ada],[bob,Bob],[cyd,Undef]],
Person_employable=[[ada,true],[bob,true],[cyd,false]],
Person_eMailAddrs=[[ada,ada@apple.com],[ada@gmail.com],[cyd,Undef_Set]],
Company_name=[[apple,Apple],[ibm,IBM]],
Job_minSalary=[[apple,1000],[ibm,1100]],
Job_salary=[[job1,2000],[job2,1200]]
```

These relation instances directly result in the class diagram instance visualized in the object diagram shown in Fig. 2.

3.3 User-Defined Search Space Configuration

For searching valid instances of relational models, Kodkod requires a restricted search space, i. e., a predetermined universe of atoms and bounds to the declared relations. Upper bounds determine the set of all possible tuples for each relation. Lower bounds, instead, declare sets of tuples which *must* occur in a valid instance, i. e., a partial solution. A comfortable way for specifying partial solutions is the translation of a partial user-defined object diagram into the lower bounds of the concerned relations. This forward translation can be done analogously to the backward translation illustrated in Sect. 3.2.

Since the search space directly influences the search efficiency of Kodkod, the aim is to minimize the upper bounds. Respective optimizations are in particular possible in the context of partial solutions, since the existence of specific tuples in

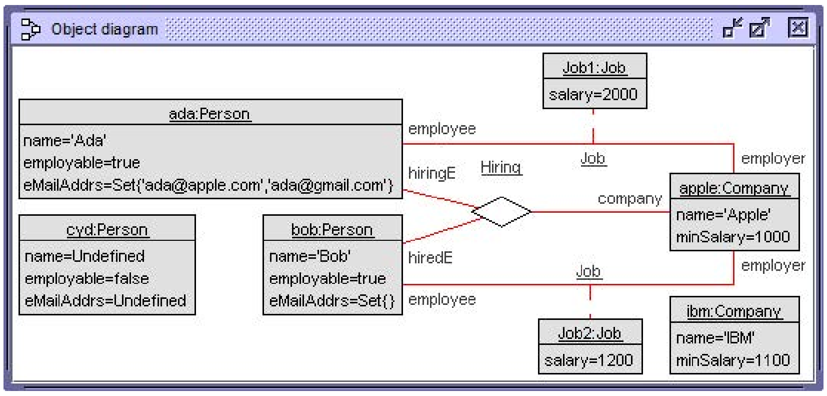

Fig. 2. Translation Result from Relation Instances to Class Diagram Instances

the lower bounds often preclude the existence of other tuples in a valid instance. Those tuples can be removed from the upper bounds, e. g., if a partial solution assigns the name `Ada` to object `ada`, the upper bounds of relation `Person_name` can be filtered with respect to tuples assigning other names to this object.

The search space configuration can be extended by relational constraints which, for example, determine the minimum and maximum numbers of defined links of a specific association, or attribute values of a specific attribute. Those properties cannot be configured by bounds, as they do not concern specific tuples.

An implementation of the considered transformation should provide means for easy configurations while hiding the particularities of relational logic, e. g., allowing the user to determine the minimum and maximum number of objects, forbidding specific links, or defining ranges of available attribute values.

3.4 From OCL Constraints to Relational Constraints

Class diagrams can be annotated with OCL invariants which constrain the set of valid class diagram instances. OCL invariants, representing Boolean OCL expressions, are transformed into relational formulas. Additionally, in our validation approach user-defined validation tasks specifying properties the searched model instance must fulfill are made available in form of temporary OCL constraints. In this section, we consider the translation of individual interesting and important OCL operations. The transformation of operations not discussed in this section can be inspected in [13].

Boolean Operations. The intrinsic transformation approach makes use of the two-valued relational logic. Consequently, Boolean OCL expressions result in relational formulas, in contrast to non-Boolean OCL expressions which result in relational expressions, i. e., relation instances. For example, consider the Boolean operation *xor* which is the only Boolean operation with no direct counterpart in relational logic:

$expr_1$ ***xor*** $expr_2 \xrightarrow{t}$ `(`$\overset{t}{\overrightarrow{expr_1}}$ `&& !`$\overset{t}{\overrightarrow{expr_2}}$`) || (!`$\overset{t}{\overrightarrow{expr_1}}$ `&&` $\overset{t}{\overrightarrow{expr_2}}$`)`, with $\overset{t}{\overrightarrow{e}}$ denoting the transformation result of OCL expression e into a relational expression or formula, respectively.

Since the Boolean values of relational formulas cannot be stored in relations, we define two relational operations (a) for mapping Boolean atoms (`true` and `false` which can occur as Boolean attribute values or as Boolean literals in OCL expressions) to relational truth values, and (b) for mapping relational truth values into atomic values:

(a) `expr2formula(e)`:`Formula` = `e=[[true]]`
(b) `formula2expr(f)`:`Expression` = `f => [[true]] else [[false]]` (if-then-else)

Operation (a) reveals that the three-valued logic of OCL is encoded into two-valued relational logic by mapping the undefined value to the value *false*. This realization can influence the validity of OCL invariants. Consider, for example, the OCL constraint $expr_1$ *and* $expr_2$ *implies* $expr_3$ which would evaluate to *Undefined*, and thus would be violated, if $expr_1$ evaluates to *Undefined* and the other expressions to *false*. The corresponding relational constraint, however, would be fulfilled. This disadvantage can be avoided by explicitly treating possible undefined values within a constraint, e. g., by applying explicit case distinctions and the OCL operation *oclIsUndefined*. As a consequence, the modeler has to be aware of situations in which an OCL expression can be undefined (which is anyway a preferable modeling style).

Integer, OclAny and Other Operations. Except for the explicit handling of the undefined value, integer operations are directly translated into their counterparts provided by relational logic. OclAny operations like *equality*, *inequality* or *oclIsUndefined* result in Boolean values, hence, requiring the application of the expression, formula mapping operations discussed before. However, their transformation is also straightforward. The distinct operations and statements *allInstances*, *let*, *if-then-else*, and the access of attribute values also yield plain relational constructs. For details see [13].

Set Operations. In the majority of cases, OCL set operations like *union*, *including*, *includes*, *forAll* or *exists* can be directly transformed into equivalent relational logic expressions or formulas, respectively. In this subsection, we consider the prominent set operation *collect* which, on the one hand, is often used for comfortably collecting specific (possibly calculated) values, on the other hand, is not handled in other works on translating OCL into relational logic. Furthermore, *collect* is implicitly applied for navigating a UML class diagram using the dot shortcut which we will consider later.

src->***collect***$(v \mid body(v)) \xrightarrow{t}$ $\overset{t}{\overrightarrow{src}}$`=[[Undef_Set]] => [[Undef_Set]] else`
`rflattenUndef(rcollect(v,`$\overset{t}{\overrightarrow{src}}$`,`$\overset{t}{\overrightarrow{body}}$`(v)))`,

where *body(v)* represents an arbitrary OCL expression in which variable v may occur, `rflattenUndef` and `rcollect` are relational operations which we have defined for transforming the OCL *collect*. The case distinction ensures that an undefined source collection (*src*) again results in an undefined collection.

The operation `rcollect` requires three arguments; a variable `v`, the translated source expression, and the translated body expression in which `v` may occur. First, this operation creates a binary relation via comprehension which relates each element of the source collection to the evaluation result of the respective body expression. For instance, the OCL expression *Set{1,2,3}->collect(i|i*i)* would yield the intermediate relation `[[1,1],[2,4],[3,9]]`. The transformation respects the fact that the result of *collect* must be flattened. If the body expression, results in a set of values, each element of the source collection is related to each element of this set via an individual tuple, i. e., the result is automatically flattened. After that, the first tuple position is cut off to obtain the desired evaluation result, e. g., with respect to the current example `[[1],[4],[9]]`:

$$\texttt{rcollect(v,}\overset{t}{\overrightarrow{src}}\texttt{,}\overset{t}{\overrightarrow{body}}\texttt{(v))} = \texttt{univ.\{v:}\overset{t}{\overrightarrow{src}}\texttt{, res:}\overset{t}{\overrightarrow{body}}\texttt{(v) | true\}}$$

The body of a *collect* expression can result in collection values which are implicitly flattened in the context of the OCL *collect*, e. g., the expression *Set{Undefined,Set{1}}->collect(i|i)* evaluates to *Bag{Undefined,1}* of type *Bag(Integer)*, while the source collection is of type *Set(Set(Integer))*. That is, undefined set-valued body expressions evaluate to an undefined value in the flattened result. For this reason, we need the operation `rflattenUndef` which checks if undefined collections (expressed by the atom `Undef_Set`) occur, and transforms them into `Undef` representing undefined single-values:

```
rflattenUndef(e) = Undef_Set in e => (e-Undef_Set)+Undef else e
```

Please note that the relational representation of *collect* always results in *sets* of values, while its OCL counterpart either results in bags or sequences, possibly yielding duplicate values and specific orders. The intrinsic approach thus restricts the expressiveness of *collect*. However, in many circumstances, not a specific order or the number of duplicate values is crucial, but the collection of distinct values. Let us consider this fact with the help of two concrete OCL invariants based on the class diagram shown in Fig. 1:

```
context c:Company
inv MinimumSalaryMaintained: c.job.salary->min() > c.minSalary
inv HiringPersonEmployed:
  c.hiringE->notEmpty() implies c.hiringE.employer->asSet()=Set{c}
```

The first invariant ensures in the context of a company the lowest paid job to yield a salary higher than the minimum salary determined by the company. The expression *c.job.salary* implicitly applies a *collect* via the dot shortcut, collecting all salaries for each job. The aim is to obtain the lowest salary. The number of employees yielding the lowest salary is irrelevant. The other invariant ensures that persons can only hire employees for their own company. Again, the only purpose of expression *c.hiringE.employer->asSet* is to collect the *distinct* employers of persons who hire for company *c*. Consequently, despite the restrictions, the intrinsical approach supports a large variety of practical models.

Navigation. Our transformation approach allows for navigating arbitrary reflexive and non-reflexive n-ary associations and association classes. We consider the general OCL navigation expression *expr.role* representing the navigation via association *assoc* from the evaluation result of *expr* (which yields a defined or undefined object), i. e., from association end i, to the *role* at association end j. For keeping the translation clear, we introduce the auxiliary operations `univ_r` and `univ_l` which represent multiple applications of universe joins from the right or the left side, respectively:

$univ\_r(\mathtt{e}, n) = if\ \ n > 0\ \ then\ \ univ\_r(e, n-1)\mathtt{.univ}\ \ else\ \mathtt{e}$
$univ\_l(\mathtt{e}, n) = if\ \ n > 0\ \ then\ \mathtt{univ.}\,univ\_l(e, n-1)\ \ else\ \mathtt{e}$
Example: $univ\_r(\mathtt{e}, 3) = \mathtt{e.univ.univ.univ}$

expr.role (via n-ary association *assoc* from association end i to end j) $\xrightarrow{t}$

$\overrightarrow{expr}^{t}$`=[[Undef]] => [[`$uv$`]] else`

if $i < j$ then $univ\_r(univ\_l(\overrightarrow{expr}^{t}\,\mathtt{.}\,univ\_l(\mathtt{assoc}, i-1), j-i-1), n-j)$

else $univ\_l(univ\_r(univ\_r(\mathtt{assoc}, n-i)\,\mathtt{.}\,\overrightarrow{expr}^{t}, i-j-1), j-1)$,
where uv is equal to `Undef_Set` if association end j is set-valued, and uv is equal to `Undef` if end j is object-valued.

Let us consider some example navigation expressions based on association *Hiring* and association class *Job* shown in Fig. 1:
apple.hiringE (from association end 3 to end 1) $\xrightarrow{t}$ `(Hiring.[[apple]]).univ`.
apple.hiredE (from end 3 to end 2) $\xrightarrow{t}$ `univ.(Hiring.[[apple]])`.
bob.company[hiredE][1] (from end 2 to end 3) $\xrightarrow{t}$ `[[bob]].(univ.Hiring)`.
ada.job (from end 2 to end 1) $\xrightarrow{t}$ `(Job_assoc.univ).[[ada]]`

As we have mentioned before, the dot shortcut, i. e., an implicit *collect*, provided by OCL allows us to easily collect objects while navigating through a class diagram, i. e., via more than association. Consider, for instance, the expression *apple.hiringE.employer* including an ordinary navigation starting from an object (*apple*), as well as an implicit *collect* based on the navigation result which further navigates to association end *employer*. This shortcut expression is equivalent to *apple.hiringE*`->`*collect(p|p.employer)*. A (complete) transformation of this expression is shown at the end of this section.

Our transformation approach allows us to differentiate between three distinctive cases which is required by OCL. (a) If *expr* within *expr.hiringE.employer* is undefined, the whole expression results in an *undefined set.* (b) If *expr.hiringE* results in a defined set including at least one unemployed person, the whole shortcut expression results in a *set including the undefined value.* (c) If *expr.hiringE* results in an empty set, the whole expression results in an *empty set.* These meaningful cases cannot be expressed by approaches like [1] due to language restrictions with respect to Alloy.

[1] Since the association is reflexive, i. e., persons can participate in *Hiring* links in different roles, the association end from which the navigation starts must be determined within brackets if ambiguous.

$$\texttt{rflattenUndef(rcollect(p,}\overset{\xrightarrow{t}}{apple.hiringE}\texttt{,}\overset{\xrightarrow{t}}{p.employer}\texttt{))} =$$

$$\overset{\xrightarrow{t}}{apple.hiringE}\texttt{=[[Undef\_Set]] => [[Undef\_Set]] else}$$

$$\texttt{(Undef\_Set in univ.}\{\texttt{p:}\overset{\xrightarrow{t}}{apple.hiringE}\texttt{, res:}\overset{\xrightarrow{t}}{p.employer}\texttt{ | true}\}\texttt{ =>}$$

$$\texttt{((univ.}\{\texttt{p:}\overset{\xrightarrow{t}}{apple.hiringE}\texttt{, res:}\overset{\xrightarrow{t}}{p.employer}\texttt{ | true}\}\texttt{)-Undef\_Set)+Undef}$$

$$\texttt{else univ.}\{\texttt{p:}\overset{\xrightarrow{t}}{apple.hiringE}\texttt{, res:}\overset{\xrightarrow{t}}{p.employer}\texttt{ | true}\}\texttt{)}, \text{ with}$$

$$\overset{\xrightarrow{t}}{apple.hiringE} =$$

$$\texttt{[[apple]]=[[Undef]] => [[Undef\_Set]] else (Hiring.[[apple]]).univ}, \text{ and}$$

$$\overset{\xrightarrow{t}}{p.employer} = \texttt{p=[[Undef]] => [[Undef]] else p.(univ.Job\_assoc)}$$

4 Related Work

While there are many important approaches in the field of UML and OCL model validation, in particular for information system validation [20], there is currently only one work following our approach to directly translating UML models into pure relational models [27]. The approach focuses on automatic resolution of model inconsistencies by translating basic class diagram concepts into relations and formulas. OCL as a whole and important UML features like n-ary associations, association classes, and undefined values have not yet been explicitly concerned.

OCLexec [12,11] makes use of Kodkod in order to generate Java method bodies by animating OCL operations constrained by OCL postconditions and invariants. In this approach, OCL expressions are translated into arithmetic expressions with bounded quantifiers and uninterpreted functions, i. e., pure integer expressions. The efficient mechanisms of Kodkod [28] are applied to transform those expressions into SAT problems. However, this approach has a loose connection to our work, since the authors of OCLexec '*do not make use of higher-level features of Kodkod such as encoding of relations*'. Thus, our transformation of UML and OCL concepts into relations and relational formulas is fundamentally different from the transformation result of OCLexec.

Our work is related to approaches which translate UML and OCL into the specification language Alloy [10] which is also based on relational logic. The so-called Alloy Analyzer transforms Alloy specifications into relational models supported by Kodkod. However, the modeling concepts provided by Alloy, e. g., signatures and fields, purposefully restrict the structure of specification components. That is, on the one hand, structures of Alloy specifications result in specific relational structures, but, on the other hand, not all relational structures supported by Kodkod can be modeled with Alloy. Consequently, several aspects of UML and OCL like the adequate handling of undefined values are not supported by Alloy, and thus are not directly realizable by approaches like UML2Alloy [1].

While UML2Alloy is an elaborated tool for validating UML and OCL models, it does not handle UML concepts like n-ary associations and association classes, or OCL operations like *collect*. The authors of CD2Alloy [18] pursue a deep embedding by defining class diagram constructs as new concepts within Alloy, enabling, for example, the comparison of two class diagrams. The work discussed in [19] aims to check the consistency between class and object diagrams by explicitly modeling object diagram concepts in Alloy. A backward transformation from original Alloy specifications into UML and OCL models is presented in [8]. The authors in [4] translate conceptual models described in OntoUML for validation purposes into Alloy.

Kodkod has been successfully applied in different fields, e. g., for executing declarative specifications in case of runtime exceptions in Java programs [25], reasoning about memory models [29], or generating counterexamples for Isabelle/HOL a proof assistant for higher-order logic (Nitpick) [3].

There are many other works concerning the validation of UML and OCL models which do not base on Alloy or Kodkod. For instance, a direct translation of UML and OCL concepts into SAT has been addressed in [26]. However, a direct translation cannot benefit from existing translation mechanisms like the sophisticated symmetry detection and breaking scheme which enables an efficient handling of partial solutions, or the detection and exploitation of redundant structures in formulas which are implemented in Kodkod. A translation of specific UML and OCL features into constraint satisfaction problems (CSP) is done in [6]. Answer set programming (ASP) [21], the constructive query containment (CQC) method [22], or rewriting-based techniques [23,7] are applied for analyzing static and dynamic model aspects. The named approaches differ from more interactive approaches like [5] involving verification by theorem proving.

5 Conclusion

In this paper we have presented the details of a bidirectional transformation from UML and OCL into relational logic and back, while focussing on the essential concepts of UML models and central OCL operations. Our so-called intrinsic approach implies restrictions at the UML and OCL side, but, on the one hand, enables the direct use of relational constructs, and, on the other hand, does still support a large variety of practically useful models.

Future work will comprise the finalization of our extrinsic approach which has been developed parallel to the current intrinsic approach. We will discuss a detailed comparison of (a) the intrinsic and extrinsic approach, and (b) our approaches and other relational and non-relational UML and OCL model validation approaches. A comparison will consider the supported UML and OCL features based on the OCL benchmark [16] as well as the efficiency with respect to models of different scale and purpose. Furthermore, the transformation will be extended regarding dynamic aspects, e. g., involving OCL pre- and postconditions, UML state machines, and sequence diagrams, and the mechanisms for specifying and optimizing the search space of model instances will be consolidated.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and System Modeling 9(1), 69–86 (2010)
2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
3. Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
4. Braga, B.F.B., Almeida, J.P.A., Guizzardi, G., Benevides, A.B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. ISSE 6(1-2), 55–63 (2010)
5. Brucker, A.D., Wolff, B.: HOL-OCL: A Formal Proof Environment for UML/OCL. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 97–100. Springer, Heidelberg (2008)
6. Cabot, J., Clariso, R., Riera, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW 2008, pp. 73–80 (April 2008)
7. Clavel, M., Egea, M.: ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 368–373. Springer, Heidelberg (2006)
8. Garis, A.G., Cunha, A., Riesco, D.: Translating Alloy Specifications to UML Class Diagrams Annotated with OCL. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 221–236. Springer, Heidelberg (2011)
9. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3), 27–34 (2007)
10. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2006)
11. Krieger, M.P., Brucker, A.D.: Extending OCL Operation Contracts with Objective Functions. ECEASST 44 (2011)
12. Krieger, M.P., Knapp, A.: Executing Underspecified OCL Operation Contracts with a SAT Solver. ECEASST 15 (2008)
13. Kuhlmann, M., Gogolla, M.: Intrinsic Relational Approach: Transformation of OCL Operations, `http://www.db.informatik.uni-bremen.de/publications/intern/IntrinsicApproachOCL2012.pdf`
14. Kuhlmann, M., Gogolla, M.: Strengthening SAT-Based Validation of UML/OCL Models by Representing Collections as Relations. In: Tolvanen, J.P., Vallecillo, A. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 32–48. Springer, Heidelberg (2012)
15. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
16. Kuhlmann, M., Hamann, L., Gogolla, M., Büttner, F.: A benchmark for OCL engine accuracy, determinateness, and efficiency. Software and System Modeling 11(2), 165–182 (2012)
17. Kuhlmann, M., Sohr, K., Gogolla, M.: Comprehensive Two-Level Analysis of Static and Dynamic RBAC Constraints with UML and OCL. In: Baik, J., Massacci, F., Zulkernine, M. (eds.) SSIRI 2011, pp. 108–117. IEEE (2011)
18. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 592–607. Springer, Heidelberg (2011)

19. Maoz, S., Ringert, J.O., Rumpe, B.: Semantically Configurable Consistency Analysis for Class and Object Diagrams. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 153–167. Springer, Heidelberg (2011)
20. Olivé, A.: Conceptual Modeling of Information Systems. Springer (2007)
21. Ornaghi, M., Fiorentini, C., Momigliano, A., Pagano, F.: Applying ASP to UML Model Validation. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 457–463. Springer, Heidelberg (2009)
22. Queralt, A., Teniente, E.: Reasoning on UML Class Diagrams with OCL Constraints. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 497–512. Springer, Heidelberg (2006)
23. Roldán, M., Durán, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST 44 (2011)
24. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. The Pearson Higher Education (2004)
25. Samimi, H., Aung, E.D., Millstein, T.D.: Falling Back on Executable Specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010)
26. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: DATE, pp. 1341–1344. IEEE (2010)
27. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 69–84. Springer, Heidelberg (2011)
28. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
29. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. In: Zorn, B.G., Aiken, A. (eds.) PLDI, pp. 341–350. ACM (2010)
30. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. The Addison-Wesley Object Technology Series. Addison-Wesley (2003)