Scalable Automatic Test Data Generation from Modeling Diagrams

Yannis Smaragdakis
Computer Science
University of Oregon
Eugene, OR 97403, USA
yannis@cs.uoregon.edu

Christoph Csallner
College of Computing
Georgia Tech
Atlanta, GA 30332, USA
csallner@cc.gatech.edu

Ranjith Subramanian
College of Computing
Georgia Tech
Atlanta, GA 30332, USA
ranji@cc.gatech.edu

ABSTRACT

We explore the automatic generation of test data that respect constraints expressed in the Object-Role Modeling (ORM) language. ORM is a popular conceptual modeling language, primarily targeting database applications, with significant uses in practice. The general problem of even checking whether an ORM diagram is satisfiable is quite hard: restricted forms are easily NP-hard and the problem is undecidable for some expressive formulations of ORM. Brute-force mapping to input for constraint and SAT solvers does not scale: state-of-the-art solvers fail to find data to satisfy uniqueness and mandatory constraints in realistic time even for small examples. We instead define a restricted subset of ORM that allows efficient reasoning yet contains most constraints overwhelmingly used in practice. We show that the problem of deciding whether these constraints are consistent (i.e., whether we can generate appropriate test data) is solvable in polynomial time, and we produce a highly efficient (interactive speed) checker. Additionally, we analyze over 160 ORM diagrams that capture data models from industrial practice and demonstrate that our subset of ORM is expressive enough to handle their vast majority.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; H.2.3 [**Database Management**]: Languages

General Terms

Design,Languages

1. INTRODUCTION

Modeling languages offer a concise way to capture design decisions and express specifications at a level of abstraction higher than concrete code. The higher level of abstraction often lends itself to automated reasoning. Nevertheless, a competing trend is that of adding more and more functionality to modeling languages, in order to bridge the gap between design and implementation. Several modern modeling languages (e.g., UML) have distinct sub-languages, some of which are highly abstract, while others are much closer to the implementation. Automatic reasoning is easy in the former case but becomes quite hard in the latter.

In this paper, we analyze a modeling notation and, in particular, its tradeoffs between expressiveness and automatic reasoning ability. Our modeling language is *Object-Role Modeling* (*ORM*) [10, 20]: a modern, popular, and powerful data modeling language. ORM is a general language for conceptual modeling, although its primary application is in the database domain. For concreteness, we use database terminology in this paper, although the results are domain-independent. The problem we want to solve is the automatic generation of test data that respect semantic constraints expressed in the model.

Producing valid test data is a problem with several applications in practice. Developers often want sample data both to verify that their specification corresponds to their understanding of the data, and to test programs as they are being developed. Generating unconstrained data is unlikely to be appropriate, however. Useful data typically needs to respect many semantic constraints: e.g., on a given table a certain field's values may be unique (i.e., the field is a key); one table's contents may be a subset of the contents of another; data values of a field may need to be in a specific range; etc. Such constraints are often concisely captured in data modeling languages. Thus, it is convenient and intuitively appealing to use a high-level model, expressed in a data modeling language, as a blueprint for creating large volumes of concrete well-formed data automatically.

Nevertheless, the problem of satisfying well-formedness constraints mechanically is often quite hard. In our case, producing sample data from ORM models (called *diagrams* in ORM parlance) is at best an intractable problem: even a simplified version of basic ORM constraints, with each table (*predicate* in ORM terminology) holding up to a single entry, makes the problem NP-hard. Checking realistic ORM constraints is typically even harder: constraints that describe relationships among different entries of a predicate typically result in a double-exponential runtime complexity. Even with small amounts of data to generate, a naive translation of the constraints into logic needs to model not just the predicate interconnections but also the contents of predicates. The result typically far exceeds the capabilities of modern constraint solving tools. In an experiment with state-of-the-art solvers we could not get a satisfying assignment for as few as 3 entities with 20 elements each.

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

Practical uses of ORM modeling, however, often only concentrate on a modest subset of the language. The main constraints on predicates encountered in practice are special forms of internal uniqueness constraints ("this field or set of fields is a primary key"), mandatory constraints ("every value of this entity needs to participate in this predicate") and subtype constraints on objects. Thus, we define an interesting subset of ORM, ORM$^-$ (pronounced "ORM-minus"), with such commonly used yet simple constraints. We define ORM$^-$ precisely and present an algorithm that a) detects errors that make a diagram unsatisfiable in polynomial time (relative to the diagram size); and b) produces large quantities of data in time proportional to the size of the data produced. (Full ORM inputs can be used with our algorithm but constraints outside the ORM$^-$ subset are ignored.)

In overview, the novel elements of our work are as follows:

- We define ORM$^-$: a subset of the ORM modeling notation with desirable properties for automated reasoning. The satisfiability (i.e., consistency) of diagrams in ORM$^-$ can be checked efficiently—both in theoretical terms (polynomial time) and in practice (sub-second runtime). ORM$^-$ represents a sweet spot in the tradeoff between automation and expressibility: we show two small variations that make the problem intractable when added to the existing set of allowed constraints. Although there are similar results for other notations, we are not aware of any such result for ORM or modeling languages straightforwardly reducible to/from ORM.
- Our ORM subset is of practical interest: the vocabulary allowed was selected after consultation with database engineers who use the ORM notation. Our constraints are precisely those used heavily in practice, perhaps suggesting that database engineers avoid too-expressive constraints. We validate the applicability of ORM$^-$ with an extensive study of ORM diagrams in real industrial use. We examine over 160 diagrams that contain some 1800 constraints in total. Of those constraints, only 24 are not expressible in ORM$^-$. The vast majority of diagrams are completely free of constraints outside the ORM$^-$ subset.

2. BACKGROUND: ORM

Object Role Modeling (ORM) is a data modeling language that attempts to model a system in terms of its objects and the roles that they play. ORM has a graphical notation and tries to capture common properties that are well-understood by database programmers. We next present briefly the most common elements of ORM. Some advanced elements are elided, in the interest of space, since they do not qualitatively affect our subsequent discussion.

For the rest of this paper, we follow the convention of calling a model specified in ORM a "diagram". When we talk of a "model" (as a noun) of the diagram, we mean a set of data that satisfy the diagram's constraints—analogously to the use of the term "model" in logic.

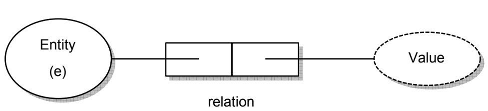

Figure 1: Basic notation

The basic components of the system being modeled are referred to as *objects*. Objects can be either *values* (i.e., well understood objects, such as a number or a string) or *entities* (derivative objects that are mapped to values). Object types are related to each other through *predicates*. Any number of object types can be related to each other, thus there can be predicates of any arity. If an object type e is related to another object type f through a predicate p, we say object types e and f play roles in the predicate p. The term *role* is used to refer to the relationship between an object type and a predicate, since an object type can be used in multiple predicates. In the relational world, a predicate can be thought of as a relation or table. The notation for value and entity types is shown in Figure 1. Value types are designated by a dotted-line ellipsis, while entity types are designated by a solid-line ellipsis. Typically each entity is identified by a single value type (e.g., a name string), which is listed in parentheses under the entity name. In general, the difference between value types and entity types has little consequence in our examples, and we will occasionally use the term "entity" to mean "entity or value".

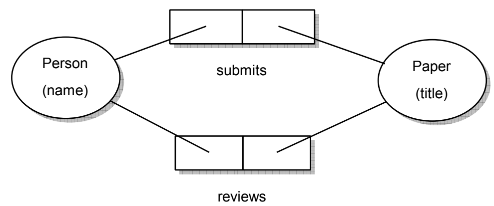

Figure 2: Simple ORM diagram

Consider the example in Figure 2. We are trying to model a simple system to store information about conference submissions. Two entity types, "Person" and "Paper" represent our objects. There are two predicates *reviews* and *submits*. Each of the entities play a role in both these predicates. A database that corresponds to this diagram will consist of two tables (assuming what is called the "standard mapping" of ORM diagrams to an implementation [11]): one with a list of people and the papers they submitted and another with a list of reviewers and papers they review.

There are no constraints specified on the ORM diagram in Figure 2. The modeler may want to specify a restriction that a person cannot both submit and review papers. Another restriction might be that a person can submit at most two papers. Or the modeler may require that a paper not have more than a set number of authors. Next, we list the main types of constraints that can be specified in an ORM diagram with examples. For a more detailed treatment of ORM, the reader is referred to Halpin's book [10].

Uniqueness constraints. This constraint specifies that the occurrence of a value in any column of a predicate is unique. In database terminology, a uniqueness constraint on a predicate identifies the keys of the corresponding table. Uniqueness can be represented in the ORM diagram by a double arrow over the roles that are unique in a predicate. For example, consider the diagram in Figure 3. The line over the role played by *Paper* in the predicate *accepted* indicates that a particular paper can occur in that table at most once—i.e., a paper can be accepted to at most one conference.

Uniqueness can span multiple roles, implying that each tuple with values from the roles is unique. Furthermore, the

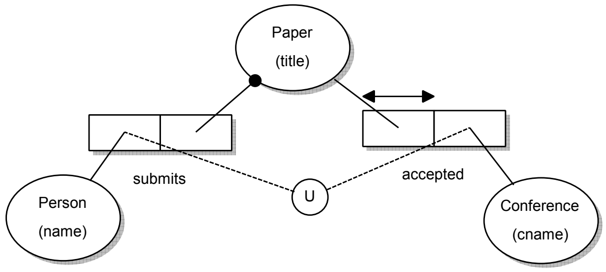

Figure 3: Uniqueness and mandatory constraints

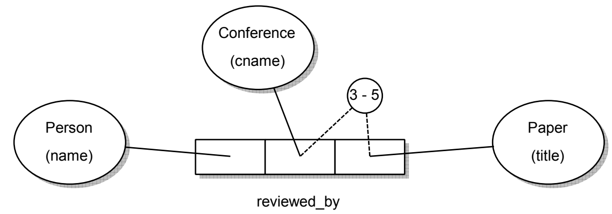

Figure 4: Frequency constraints

roles can belong to different predicates—a constraint called *external uniqueness*. In this case, the tuple spanning multiple roles is considered on the (implicit) predicate resulting from joining the actual predicates on their common values. For instance, in Figure 3, the roles for *Person* in predicate *submits* and for *Conference* in predicate *accepted* are connected by an external uniqueness constraint, signified by the circled "U" in the diagram. This specifies that no person can have more than one paper in the same conference.

In our formulation of ORM, we enforce the common implicit uniqueness constraint spanning all roles of a predicate. That is, a predicate cannot contain two tuples that are the same across all roles.

Mandatory constraints and independence. In Figure 3, the black dot connecting the entity *Paper* to the role it plays in the *submits* predicate denotes a *mandatory constraint*. This indicates that all papers in our universe are submitted.

A more complex form of the mandatory constraint is the *disjunctive mandatory constraint*. This links roles of the same entity in multiple predicates and signifies that all objects in the entity *have* to be a part of *some* (possibly all) of those predicates. This constraint is represented by connecting each of the roles that are mandatory to a black dot. We will not encounter an example of such a constraint in our discussion, except in its implicit form. In an ORM diagram, every object of an entity is implicitly assumed to participate in some of the predicates in which the entity has a role. This means that every entity in the system has an implicit disjunctive mandatory constraint over all its roles. (If an entity plays only one role, then this is equivalent to a regular mandatory constraint on that role. For instance, a black dot would be redundant on the links from entities *Person* or *Conference* in Figure 3.)

An exception to this rule is *independent* entity types, designated with an exclamation mark after their name. Objects of an independent type can exist without participating in predicates.

Frequency constraints. Frequency constraints generalize uniqueness constraints, by allowing each tuple to occur a number of times, instead of just once. In Figure 4, the frequency constraint of (3-5) on the roles played by *Conference* and *Paper* states that each combination of conference and paper that appears in the predicate has to appear three to five times—i.e., each paper needs three to five reviewers for the same conference.

Subset, Equality constraints. A *subset* constraint can be applied over two ranges of roles in two predicates. The constraint specifies that the tuples in the roles of the first predicate have to be among the tuples in the roles of the second. An important restriction here is that the types of the roles of the sub-predicate in a subset constraint have to match the corresponding roles of the super-predicate on a role by role basis. The *equality* constraint is a two-way subset: it forces the two sets of tuples to be equal, as the name suggests.

In the example of Figure 5, the subset constraint spans the entire *accepted* predicate, specifying that a paper needs to have been submitted to the conference, if it is to be accepted.

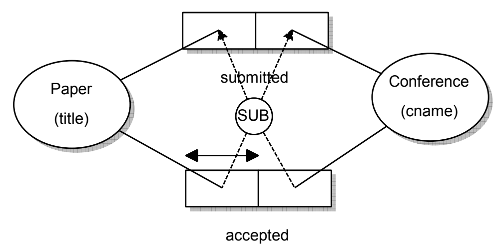

Figure 5: Subset constraint

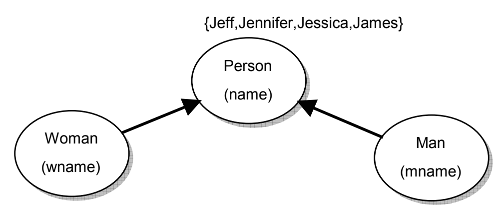

Figure 6: Subtype and value constraints

Subtype constraints. This constraint is used to model usual subtyping situations. For example, the *Man* and *Woman* entity types can be declared as subtypes of *Person*, as shown in Figure 6.

Value and cardinality constraints. A value constraint is used to enumerate the range of values that an entity type may have. In the example of Figure 6, the allowed values for the *Person* entity type are given explicitly. Value constraints can also be used with a range notation—e.g., one can specify that the *Age* value type needs to be a number between 1 and 150. It is important to make the distinction between a value constraint, which specify the allowed values in a type, and *cardinality* constraints, which specify how many elements a type has. Clearly a value constraint implies an upper bound on the cardinality. Our graphical notation does not reflect cardinality constraints, although they do exist in the diagram. A cardinality constraint can be either a number or a range.

Exclusion constraints. This constraint is used to model a situation where a particular value can occur in only one table in any valid model of the diagram. For instance, in a system that stores the *current* state of papers, a paper cannot be both submitted and accepted. We will see the graphical notation for exclusion constraints in Section 3.1.

Ring constraints. A predicate represents a relation over the sets of values for each of the entities playing roles in the predicate. For predicates with all roles played by the same entity, we can specify various properties that the relation should possess. For example, in Figure 7, we state that the *works_with* predicate is *symmetric*. That is, if a person A works with a person B, then B also works with A.

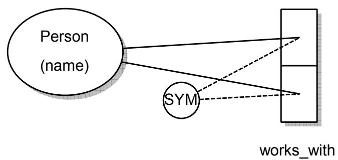

Figure 7: Ring constraint

Other types of ring constraints include *acyclic*, *transitive*, *intransitive*, *reflexive*, *irreflexive*, and *asymmetric*.

3. DIFFICULTY OF ANALYZING ORM

The problem we want to address is that of generating sample databases from ORM diagrams. We assume that we receive as input an ORM diagram and we want to determine its satisfiability (i.e., the existence of data that populate all types and predicates and satisfy all constraints). Some variants of ORM (including the one supported by the tool used in our experiments) allow arbitrary query-like code (in Datalog or SQL) to be used as a notation for constraints,[1] making the problem of determining satisfiability undecidable. (Although query languages guarantee termination, other common problems, such as query subsumption, are undecidable.) Even without allowing arbitrary queries, undecidability is not uncommon when dealing with highly expressive integrity constraints. For instance, the general class of "numeric dependencies" [7] in databases has no decidable inference process.

Even if we limit ourselves to "standard" ORM constraints, however, the problem is at least NP-hard. We next discuss the difficulties involved, and present a (failed) brute-force attempt at the problem, which serves to frame our later restriction of ORM to a simpler subset.

3.1 NP-Hardness

Our problem of producing satisfying data for ORM diagrams is NP-hard. Bommel et al. have shown [23] the NP-hardness of some slightly different variants of the ORM satisfiability problem. Nevertheless, it is easy to produce a much simpler NP-hardness proof for our exact formulation of ORM and the satisfiability problem.

We reduce the well-known NP-complete boolean satisfiability problem, 3SAT, to our problem. A 3-CNF formula has the form $(a_1 \vee a_2 \vee a_3) \wedge (b_1 \vee b_2 \vee b_3) \wedge ...$, with $a_1, a_2, a_3, b_1, ...$ representing expressions q_i or $\neg q_i$ over a set of boolean variables $q_1, q_2, ..., q_n$. A CNF formula can be translated to an ORM diagram using the following steps.

1. Introduce an object type for each variable q_i in the CNF formula
2. Introduce an object type for each clause c_i in the CNF formula
3. Impose a value constraint on each type introduced above. The value constraint restricts each type q_i to have just one value 'qi'. Similarly, each clause is restricted to have one value 'ci'.
4. For each variable q_i, let n be the number of times it occurs in the clauses c_i to c_j. If $n > 0$, create a $(n+1)$-*ary* predicate q_i-*true* for the variable q_i.
5. Similarly, for each variable $\neg q_i$ let m be the number of times it occurs in the clauses c_i to c_j. If $m > 0$, create a *(m+1)-ary* predicate q_i-*false* for the variable $\neg q_i$.
6. Connect type q_i to predicates q_i-*true* and q_i-*false*
7. Impose an *exclusion constraint* on the roles played by q_i in q_i-*true* and q_i-*false*
8. Connect each clause c_i to the predicate q_i-*true* if it contains q_i or q_i-*false* if it contains the variable $\neg q_i$

Consider a CNF formula $(q_1 \vee q_2 \vee \neg q_3) \wedge (q_1 \vee \neg q_2 \vee q_3) \wedge (\neg q_1 \vee q_2 \vee \neg q_3) \wedge ...(...)$. Figure 8 illustrates the relevant part of the formula's transformation.

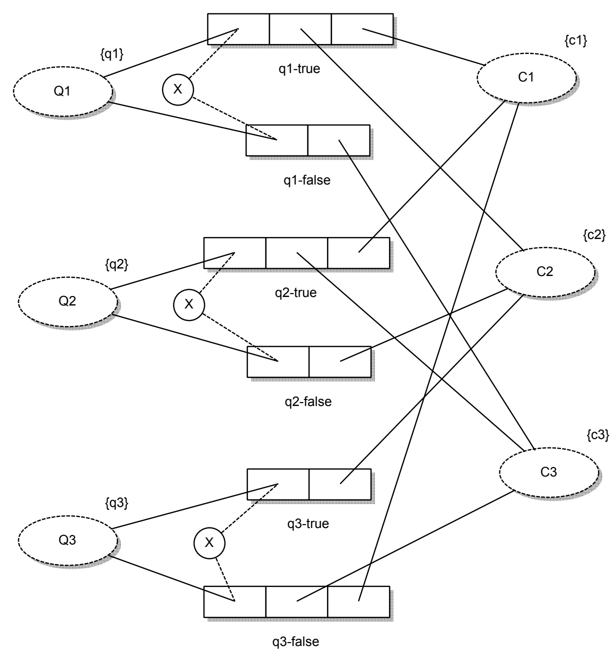

Figure 8: 3SAT to ORM

Based on the semantics of ORM constraints, it is easy to see that the 3SAT formula is satisfiable iff the ORM diagram is satisfiable. The exclusion constraints on the roles played by the q_is force each value to appear in at most one of the q_i-*true* or q_i-*false* predicates. Recall that there is an implicit disjunctive mandatory constraint over all the roles played by the same type, since none of the types are "independent". Therefore, each q_i value appears exactly once

[1]The full set of ORM constraints is not standardized ("not all versions of ORM support all these symbols" [9]). The constraint that allows us to write queries as part of the diagram is the asterisk ("*") constraint—symbol 23 in [9].

in the predicates. A similar implicit disjunctive mandatory constraint on the roles played by the c_is ensures that the single value of each of these entities appears in at least one of the predicates, effectively implementing a disjunction.

Note that the ORM satisfiability problem is NP-hard, but not clearly NP-complete. Bommel et al. claimed an NP-completeness proof for their two satisfiability problems [23]. The proof is based on a brute-force nondeterministic guess of the contents of each predicate. The claim is that because all populations are bounded by m—the maximum frequency constraint upper bound—one can nondeterministically choose populations with up to m elements for each entity and predicate in polynomial time. Yet the populations will be of up to size m, which is *exponential* relative to the *representation* of m, which is the input to the problem. Thus the argument of Bommel et al. is wrong: even a non-deterministic Turing machine cannot guess all possible populations in time polynomial to the size of the input. (We reported this finding to the authors in May 2006.)

We speculate that when complex constraints are included (such as subset or ring constraints) the problem is undecidable (or with a double-exponential complexity, if there is a known size bound). Yet, for a simpler subset the problem is just NP-complete, and for a yet simpler subset it is polynomial, as we discuss in Section 4.

3.2 Brute-Force Approach

As we saw, the ORM diagram satisfiability problem is fairly hard. There are, however, good reasons to try a combination of brute-force translation and heuristic approaches. There is a standard translation of ORM constraints to first-order logic (e.g., [8]) that directly captures the semantic intricacies of constraints. At the same time, reasoning tools have matured and state-of-the-art constraint solvers and SAT solvers often achieve impressive scalability. In practice, we found the brute-force + heuristic approach to be a bad fit for our problem. Nevertheless, the result yields insights that suggest a scalable solution.

We translated ORM diagrams to inputs for two different tools: the CoBaSa tool [17] and the Alloy Analyzer [13, 21]. This is a good choice of candidate tools: Alloy is well-known and mature, yet does not emphasize scalability, while CoBaSa offers an expressive front-end, an efficient translation, and a state-of-the-art pseudo-boolean constraint solver (PBS [1]) as its back-end. We used the standard translation of ORM to first-order logic, customized to the needs of the tool at hand. Generally we map every ORM predicate to a logic predicate, every value of an entity to a logical value, and specify constraints using a logic notation. For instance, if there is a mandatory constraint on the role played by the entity A in a predicate, this would translate to CoBaSa as:

```
For_all a in A { Sum i in id tableA(i, a) >= 1 }
```

This states that every value in the domain of A has to occur at least once in *tableA*.

The results of one of our CoBaSa experiments are shown in Table 1. We use an input with three entity types: A, B, C. These are linked with a single ternary predicate (A,B,C) and a frequency constraint spanning (B,C). Furthermore, B and C are mandatory. This simple input should yield a lower bound for the cost of adding more complex constraints.

Overall, this approach does not scale very well. Even our simple, small examples take minutes to process. The problem size (in terms of variables and clauses) scales exponentially relative to the original input. (Note that the original input is the logarithmic representation of the number n of entities, while the input to the solver is $O(n^c)$ clauses—in this case $c = 4$.) We certainly cannot answer the question of whether a satisfying configuration exists for realistic inputs (which will specify that entities contain millions of objects: recall that we want to produce test databases that satisfy the stated constraints). Our experience with the Alloy Analyzer was quite similar: we found it unable to yield solutions for an example domain of 3 entities with 5 to 40 values each.

Table 1: Relation (A,B,C) with frequency constraint of (min=2, max=3) spanning (B,C). B and C are mandatory. PBS aborts SAT solving after 1500 seconds. The first four columns give cardinalities. "vars" and "clauses" measure the complexity of the SAT problem. SAT indicates a satisfiable problem. The last column is the CoBaSa processing time.

A	B	C	pred	vars	clauses	SAT	time [s]
3	2	2	10	150	200	yes	0.23
3	2	2	15	225	300	no	197.92
10	2	2	12	264	240	yes	0.12
10	2	2	13	286	260	no	119.51
10	2	2	15	330	300	no	260.67
10	4	4	40	2000	3200	yes	0.90
10	5	5	25			no	abort
10	5	5	70	4900	8750	yes	3.84
10	6	6	100	9400	18000	yes	4.76
10	8	8	150	23100	48000	yes	22.50
10	10	10	200	46000	100000	yes	68.14
10	10	10	250	57500	125000	yes	180.10

The problem is that the brute-force approach is not a very good fit for our problem. It is much better for dealing with logically deep constraints, than with size constraints on a large space. Nevertheless, the approach has value for detecting some unsatisfiable (i.e., contradictory) configurations. It is often the case that a small model is sufficient for disproving the consistency of constraints. This has been a standard argument for practical uses of Alloy. For this to apply to our problem, the original ORM diagram must contain no explicit size constraints on object types or role frequencies that will make the search space unmanageable. A good example is a diagram that has no frequency, value, or cardinality constraints but has a predicate with two ring constraints, one irreflexive, one transitive, and two mandatory constraints on the same roles. This is a logically unsatisfiable constraint system: a transitive relation on a finite space will eventually include either a limit element or a cycle. The limit element cannot occur on the right side of the relation (so the mandatory constraint is violated), and the cycle (plus transitivity) contradicts the irreflexivity. This deep-but-not-size-sensitive reasoning is handled well by a brute-force translation into input for constraint solvers.

Although the brute-force approach fails, it suggests the direction to proceed. The problem with the brute-force approach is that it entails an inherent blowup in the problem size, because it models the interrelations of the *members* of types. This makes the problem exponentially harder: for an input of N bits, the possible values are up to 2^N and there are up to 2^{2^N} possibilities for the contents of entities and predicates. For complex constraints, such as ring, subset, and exclusion, this modeling seems necessary. For other constraints, however, we only need to concern ourselves with the *sizes* of types. Thus, if we limit ourselves to simple constraints that only deal with the sizes of sets and not their contents, we can obtain a scalable solution. Fortunately, these simple constraints turn out to be exactly the ones used overwhelmingly in practice.

4. ORM⁻: AN EFFICIENT AND EXPRESSIVE SUBSET

Given the difficulty of producing concrete data from full ORM specifications, we concentrated on a realistic subset of ORM. This subset was arrived upon through continuous refinement over the course of several months, in consultation with engineers of LogicBlox—a company that markets a database engine and ORM tools. The end result is a subset of ORM that captures the most commonly used constraints in practice, yet without sacrificing the ability to generate data very efficiently. We call our ORM subset ORM^- (pronounced "ORM minus").

4.1 Definition of ORM⁻

ORM^- is the ORM subset with the following constraints:

- *Uniqueness* constraints: only internal uniqueness (i.e., over roles in a single predicate) is supported. Uniqueness constraints can span multiple roles, but no two uniqueness constraints can overlap. E.g., it is not expressible that, in a single predicate, both roles A/B and roles B/C form a unique key.
- *Mandatory* constraints on a single role. Disjunctive mandatory constraints are not allowed, although the standard implicit disjunctive mandatory constraint holds, over all roles played by the same (non-independent) type. Independent types are also supported.
- *Frequency* constraints, which, just like uniqueness constraints, cannot overlap.
- *Value and cardinality.*
- *Subtype* constraints. We assume that a subtype's value constraints specify a subset of those of the supertype.

Notably absent are subset constraints, exclusion constraints, and ring constraints. It is possible that some of those can be added while maintaining the desirable properties of ORM^- (i.e., sound and complete satisfiability decision in polynomial time).[2] Some additions (e.g., exclusion) immediately result in an NP-hard problem, however.

4.2 Testing Satisfiability for ORM⁻

A first question regarding satisfiability of ORM^- is "what are instances of *un*-satisfiability?" Diagrams in ORM^- can be unsatisfiable for a variety of reasons. Figure 9 shows three examples.

The top example represents a simple instance of contradictory cardinality constraints. (Recall that cardinality constraints are not depicted graphically in our diagrams, although they are maintained in the properties of diagram

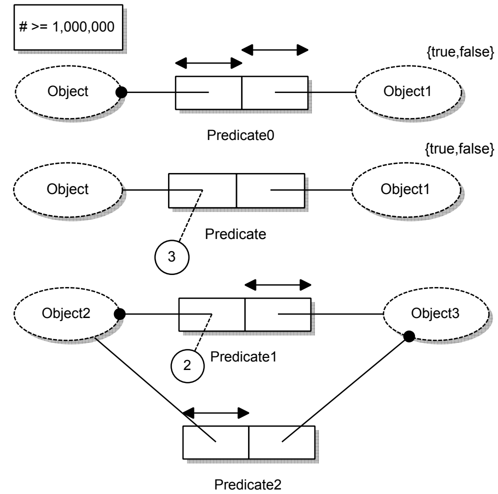

Figure 9: Three unsatisfiable diagrams.

elements. In this case, we show the constraint in a text comment.) The diagram cannot be satisfied since it requires at least a million objects of one type (the designer wants a large test database) but the type is in a one-to-one mapping with a subset of a two-object type. The middle diagram cannot be satisfied since the predicate needs to contain three tuples for each left-hand-side value, yet the right-hand-side type has at most two values. This requirement violates the implicit constraint that all tuples in a predicate be unique. The bottom diagram is a bit more interesting: through a cycle of constraints the cardinality of type *Object2* is required to be at least two times itself.

We can test the satisfiability of ORM^- diagrams in polynomial time. The key property of ORM^- is that all its constraints can be expressed as *numeric constraints* (inequalities) on the sizes of sets, instead of their contents. The sets are polynomial in number, relative to the original input. Specifically, we translate all ORM^- constraints into numeric constraints of the form $c \cdot x \leq y_1 \cdot y_2 \cdot ... \cdot y_n$, or $x \leq y_1 + y_2 + ... + y_n$, where c is a positive integer constant and x, y_1, ..., y_n are positive integer constants or variables. Note that, although these are constraints on the integers (suggesting that the problem is hard), we have no addition or multiplication of *variables* (i.e., unknowns) on the left hand side of the inequality.

Specifically, the translation rules are as follows:

- For each type A, introduce a fresh variable a, representing its cardinality. (We follow this convention of upper and lower case letters in the following.) For each predicate R, introduce a variable r, representing its cardinality (number of tuples—distinct by definition). For each role in R played by type A, introduce a variable r_a, representing the number of unique elements from A in this role.
- Produce inequalities $r_a \leq a$, and $r_a \leq r$, for each role that type A plays in predicate R.
- For each *cardinality* constraint that limits the cardinality of a type A to be in a range *min...max*, produce inequalities $min \leq a$ and $a \leq max$. Similarly for predicates. We assume, without loss of generality, that each type and predicate has a maximum cardinality constraint. (They already have a minimum cardinality constraint of 1, as we

[2]For instance, we speculate that with a more complex translation we can support external uniqueness constraints, overlapping frequency constraints, as well as subset constraints under the restriction that all other constraints on the superset predicate are identical to the constraints of the subset predicate. We have not, however, attempted to prove that such a generalization works.

do not want empty types or predicates in the solution.) In practice, for types or predicates that do not have an explicit cardinality constraint, we can produce the numeric constraint $a \leq M$, where M is an upper bound of the size of all entities or of the desired output (e.g., MaxInt).

- For each *value* constraint on a type A, produce inequality $a \leq ds$, where ds is the domain size defined by the value constraint. E.g., for a value constraint that restricts a type to four distinct values $ds = 4$. For a value constraint of the form 5...30, $ds = 26$.
- For each *mandatory* constraint on the role played by type A in predicate R, produce inequality $a \leq r_a$. (Since we also have $r_a \leq a$ for each role, the result is an equality, but all our reasoning is done in the inequality form.)
- For each *frequency* constraint, with frequency range $f_{min}...f_{max}$, on the roles played in predicate R by types A, B, ..., K, introduce a variable $r_{ab...k}$, representing the number of unique tuples in these roles. Produce the following inequalities:

  $r \leq f_{max} \cdot r_{ab...k}$
  $f_{min} \cdot r_{ab...k} \leq r$
  $r_{ab...k} \leq r_a \cdot r_b \cdot ... \cdot r_k$
  $r_a \leq r_{ab...k},\ r_b \leq r_{ab...k},\ ...,\ r_k \leq r_{ab...k}$

  Note that we are representing the number of unique tuples only for sets of roles that have a frequency constraint, not for all subsets of the roles of a predicate. Note also that uniqueness constraints and constant frequency constraints are a special case of the above. (A uniqueness constraint is a frequency constraint with a minimum and maximum frequency of 1.)
- For each *subtype* constraint between types A and S, produce the inequality $a \leq s$.
- For each predicate R with roles played by types A, B, ..., N, express the implicit uniqueness constraint over all roles as follows: Produce inequality $r \leq r_{ab...k} \cdot r_{lm...p} \cdot ... \cdot r_n$, where the elements of the right hand side are all the variables corresponding to the ranges of roles that participate in some frequency constraint on the predicate, followed by all the variables corresponding to roles that are under no frequency constraint. Recall that frequency constraints cannot overlap.
- For each non-independent type A that plays roles in predicates R, S, ..., V, introduce the numeric constraint $a \leq r_a + s_a + ... + v_a$. This captures the implicit disjunctive mandatory constraint over all roles played by a type: each object of a non-independent type needs to appear in some predicate.

For illustration, consider the translation of the top example of Figure 9. If we call the left type in the diagram "A", the predicate "R", and the right type "B", and follow our naming convention for numeric variables, we get the inequalities: $1000000 \leq a$ (cardinality on A), $a \leq r_a$ (mandatory on A), $r \leq r_a$ (uniqueness on role of A), $r \leq r_b$ (uniqueness on role of B), $b \leq 2$ (value on B), as well as the implicit constraints on roles: $r_a \leq a$, $r_a \leq r$, $r_b \leq r$, $r_b \leq b$, and the implicit cardinality constraints that make every variable at least 1 and smaller than a constant upper bound. (We omit the implicit uniqueness over all roles, and the implicit disjunctive mandatory constraint, as they are redundant for this example. The implicit mandatory constraint cannot be used, anyway, unless we know that the elements shown constitute the whole diagram and not just a part of it.) It is easy to see that these numeric constraints imply $1000000 \leq 2$—a contradiction.

The above translation supports the key properties of ORM^-: The translation is sound and complete, in that an ORM diagram is satisfiable if and only if the integer inequalities resulting from the translation admit a solution. Furthermore, testing this property can be done in polynomial time. Note that the above does *not* mean that every dataset whose type, role, and predicate cardinalities satisfy the integer inequalities will satisfy the ORM constraints! Instead, it means that for every satisfying assignment of the integer inequalities, we can produce *some* dataset that will satisfy the ORM constraints. We next present in brief the key arguments establishing these properties.

Sound and Complete Decision. If the ORM constraints are satisfiable, then the numeric constraints also hold: the numeric constraints just capture size properties of sets of values that satisfy the ORM constraints. The inverse direction is harder, but our process of showing this also yields a way to produce objects that satisfy the ORM constraints. Starting from an integer variable assignment satisfying all inequalities, we can generate data for a database that satisfies the original ORM constraints, as follows:

- For each type A without a supertype, create a new objects from the set specified by the type's value constraint (if one exists). (We follow our lower/upper case convention—i.e., a is type A's cardinality in the solution of the size inequalities.) The crucial observation is that ORM^- constraints do not restrict *which* of these objects participate in a role, predicate, or type, except in three cases: a subtype should only contain objects from its supertype; the objects in a role R_A on predicate R played by type A should be a projection over R_A of all the tuples of any range of roles that includes R_A, including the entire R; the implicit disjunctive mandatory constraint should hold: each object of a type should participate in one of the roles. We can satisfy these requirements with an appropriate choice of objects. We next discuss enumerations of objects without specifying in detail how the enumeration is implemented, as long as it is clear that one exists. All enumerations mentioned can be done in time proportional to the size of the output, but some of the algorithms are fairly lengthy.
- For each subtype B of a type that has been populated with objects, pick b objects from the supertype and populate the subtype with them. Repeat until all types are populated. The ORM subtype constraint of the original diagram is now satisfied, as are any value and cardinality constraints on types.
- For each type A, populate all roles R_A, S_A, ..., V_A that A plays with r_a, s_a, etc. elements. If A is not independent, ensure that all elements of A participate in some role. This is possible since $a \leq r_a + s_a + ... + v_a$ After this step, the implicit disjunctive mandatory and any explicit mandatory constraints of the ORM diagram are satisfied.
- At this point all roles are populated with the unique objects they contain. (I.e., we have computed the projection of each predicate on each role.) For each range of roles R_A,

R_B, ..., R_K that participate in a frequency (or uniqueness, as a special case) constraint, pick $r_{ab...k}$ distinct tuples so that their projections on roles are the computed population of R_A, R_B, etc. Such tuples are guaranteed to exist since $r_{ab...k} \leq r_a \cdot r_b \cdot ... \cdot r_k$.

- Finally, for each predicate produce its entire contents by enumerating the unique combinations of the sub-tuples from each range of roles that participates in a frequency constraint, and the objects in the roles that are yet unfilled (i.e., roles that do not participate in frequency constraints) until producing r tuples. This is guaranteed to be possible since $r \leq r_{ab...k} \cdot r_{lm...p} \cdot ... \cdot r_n$. The enumeration of tuples should be done so that the sub-tuples from roles in frequency constraints are covered evenly (i.e., two sub-tuples of the same role range are used either the same number of times or with a difference of one) and all objects from roles that are not under frequency constraints are used. This is guaranteed to produce the right frequencies for the contents of ranges of roles used in frequency constraints, since for each such range R_A, ..., R_K, we have $r \leq f_{max} \cdot r_{ab...k}$, or $r/r_{ab...k} \leq f_{max}$, and similarly for f_{min}. That is, in producing r complete tuples, we are guaranteed to repeat each sub-tuple in role range R_A, ..., R_K at least f_{min} and at most f_{max} times.

Polynomial Testing. We can test the satisfiability of numeric inequalities produced by our translation through a fixpoint algorithm. Recall that our constraints are of the form $c \cdot x \leq y_1 \cdot y_2 \cdot ... \cdot y_n$, or $x \leq y_1 + y_2 + ... + y_n$, where c is a positive integer constant and x, y_1, ..., y_n are positive integer constants or variables.

1. We first rewrite all inequalities of the form $c \cdot x \leq y_1 \cdot y_2 \cdot ... \cdot y_n$ into $x \leq (y_1 \cdot y_2 \cdot ... \cdot y_n)/c$.
2. For each variable, we can maintain its current upper bound (recall that we assume cardinality constraints for all entities and predicates in the original input, although these can be MaxInt) and, on every step, we substitute the current upper bounds for all variables on the right hand side of each inequality and perform the arithmetic operations (the division is integer division, performed after all multiplications). This produces candidate upper bounds for variables. The minimum of the candidate upper bounds for a variable and its current upper bound becomes the variable's upper bound for the next step.
3. Step 2 repeats until either an upper bound crosses a lower bound (i.e., we get $x \leq k$ and $m \leq x$, where $m > k$ for some variable x) or all upper bounds remain unchanged during a step. The former signifies an unsatisfiable case, the latter a satisfying assignment.

This algorithm is correct because it maintains a conservative upper bound on cardinalities on any step. Any satisfying assignment will need to have values at most equal to the respective upper bounds. When the algorithm reaches fixpoint, all inequalities are satisfied by the current upper bounds, which, thus, constitute a solution.

It is easy to see that this algorithm runs in polynomial time relative to the size of the input. The fixpoint computation runs a polynomial number of times between successive applications of an inequality of the form $x \leq (y_1 \cdot y_2 \cdot ... \cdot y_n)/c$. (Inequalities of other forms can only apply a polynomial number of times. If there are n variables in the system, each inequality can yield a different upper bound at most n times: the variable on its right hand side with the smallest current bound can never get any smaller, as it satisfies all other inequalities in whose left hand side it appears. Similarly, the second-smallest-upper-bound variable will never get smaller after one round, etc.) Hence, the only reason that candidate upper bounds may keep becoming smaller is inequalities that contain the division operator. Yet each of those can apply at most a number of times logarithmic relative to the value of the original upper bounds (since it reduces the upper bound by a multiplicative factor) which is polynomial relative to the input size.

A polynomial solution to this problem exists only because there is no addition or multiplication of variables on the left-hand side of an inequality (i.e., all our arithmetic operations are on upper bounds, yielding a monotonicity of the problem). Interestingly, simple additions to the ORM$^-$ constraints result in addition or multiplication on the left-hand side, and make the problem NP-complete. For instance, adding exclusion constraints corresponds to variable addition on the left-hand side. Adding a constraint that requires all combinations of objects (cross product) of two types to appear in a single predicate results into multiplication of variables on the left-hand side. It is easy to show that either problem is NP-complete. (The proofs are presented in the extended version of the paper, in the interest of space.)

5. ORM DIAGRAMS IN PRACTICE

To validate the usability of ORM$^-$, we examined the use of ORM in practice by looking at example diagrams provided by our industrial partner, LogicBlox Inc. The diagrams model a range of consulting, internal use, and benchmarking projects: a retail prediction application, a cell bandwidth prediction application, a standard database benchmark (TPC-H), and various internal tools. These diagrams are the whole set of ORM data that LogicBlox has available: no selection or other filtering took place. There are, however, threats to the representativeness of the data: the majority of the diagrams were developed by a single engineer, and all diagrams model implementations that use the same database back-end. Nevertheless, given the variety of diagrams and domains, as well as the input we received in personal communication with multiple developers, we believe our study and findings to be highly valid.

Table 2 summarizes the elements of our ORM diagrams. There are 5 distinct projects, each with one or more documents, which in turn contain one or more diagrams. Each row represents a document and clusters of rows represent a project. For instance, the first project contains a single document with 7 diagrams, while the second project contains 9 documents, the largest of which has 46 diagrams. Diagrams in the same document are just different views: they can share entities and predicates, while omitting detail that is unnecessary for the aspect currently being modeled.

Overall, we analyzed 168 diagrams with about 1800 constraints. The vast majority of constraints are in the ORM$^-$ subset. Only 24 constraints are not supported by ORM$^-$. Of these, 13 are *user-defined* constraints: they add arbitrary (query) code to the high-level diagram. This means that the developer adds consistency code (e.g., "this predicate is the join of two others over these roles") to be generated together with normal consistency code that the ORM

Table 2: Elements found in ORM diagrams in practice. Dgr:# diagrams in the document. Ent:# entity types. Val ctr:# value constraints (on entity and value types, separately). IsA:# entity types that have a super type. Indp:# independent entity types. Val:# value types. Ent ref:# entity references. Rel cont:# relation container entities. Rel:# relations (predicates). Mnd:# mandatory constraints. Unq int:# internal uniqueness constraints. Frq:# frequency constraints. Unq ext:# external uniqueness constraints. Eq:# equality constraints. Ring AC:# acyclic constraints (ring). Usr:# user-defined constraints, not counting comment-only constraints. We did not see any other ORM predefined constraints such as irreflexive, intransitive, asymmetric, etc.

	Dgr	Ent	Val ctr	IsA	Indp	Val	Val ctr	Ent ref	Rel cont	Rel	Mnd	Unq int	Frq	Unq ext	Eq	Ring AC	Usr
co	7	21	0	0	0	17	2	0	0	57	0	56	0	0	0	0	4
la	11	19	3	2	3	0	0	8	0	71	7	64	0	0	0	0	0
lb	11	31	12	4	12	15	0	0	0	55	6	55	0	0	1	3	0
ld	7	6	0	1	0	1	0	15	1	76	0	73	0	0	0	0	0
le	1	3	0	1	0	0	0	2	0	5	4	4	0	1	0	0	0
ll	12	24	7	11	6	1	0	10	0	50	20	49	n 0	0	0	0	0
lol	12	32	6	10	5	0	0	12	0	112	0	111	0	0	0	0	0
lor	4	23	2	14	2	0	0	5	0	44	6	45	0	0	0	1	0
ls	1	4	0	0	0	0	0	5	0	17	0	17	0	0	0	0	0
lu	46	94	48	31	47	0	0	12	1	528	1	523	0	0	0	0	1
mc	1	2	0	0	0	3	0	0	0	4	0	4	0	0	0	0	0
md	1	1	0	0	0	0	0	2	0	4	0	4	0	0	0	0	0
mf	1	5	1	1	0	1	0	0	0	4	8	4	0	0	0	0	0
mm	3	9	0	2	0	5	0	0	0	16	8	16	1	1	1	2	1
m	1	2	0	0	0	1	0	0	0	3	0	3	0	0	0	0	1
mo	5	22	3	13	4	0	0	13	0	65	6	66	0	0	0	1	0
mp	2	2	1	0	1	0	0	5	0	20	0	15	0	0	0	0	0
mre	3	4	0	0	0	11	1	0	6	25	10	15	0	0	0	0	0
mru	2	2	0	0	0	0	0	7	0	18	0	14	0	0	0	0	0
mu	9	8	6	0	0	0	0	27	0	84	0	48	0	0	0	0	0
mv	3	9	7	0	7	0	0	10	0	45	0	35	0	0	0	0	0
ro	13	24	0	0	0	11	1	13	0	83	0	75	0	0	0	0	5
sb	1	5	4	0	4	2	0	0	1	3	0	3	0	0	0	0	1
sl	1	3	0	0	0	2	0	0	0	4	0	4	0	0	0	0	0
st	10	11	1	0	1	22	0	0	1	64	35	63	0	0	0	0	0
Total	168	366	101	90	92	92	4	146	10	1457	111	1366	1	2	2	7	13

editor produces (e.g., for uniqueness, frequency, value, or mandatory constraints). Checking the satisfiability of such user-defined constraints seems unlikely, as this is an undecidable problem for the general query language. Thus, we do not believe that there is significantly more benefit to get over what ORM$^-$ already achieves. A promising direction may be acyclicity ring constraints, which comprise 7 of the remaining 11 constraints that ORM$^-$ does not handle.

Our analysis did not find interesting errors in the ORM documents. There were 19 cases of unsatisfiability, but all were relatively benign: an entity or value was not connected to any role or supertype. Nevertheless, finding no errors is hardly surprising for this dataset since the ORM tool generates consistency checking code (in Datalog) automatically. Therefore, mistakes in the specification are overwhelmingly likely to be caught when the database is populated with real data, and the diagrams we examined have produced databases that are in real use.

The run-time of our satisfiability check was negligible. Satisfiability checking is done on a per-document basis and on a 2 GHz AMD Athlon 64 X2 machine checking the largest document (with 528 predicates) takes about 35ms. Thus, satisfiability checking is fast enough to be done interactively inside the ORM editor, while the user is editing diagrams. Test data generation takes time proportional to the output, however, and thus needs to be done offline.

The tendency we observe is for developers to prefer to encode complex constraints in code, rather than using the semantically complex constraints of the modeling language. The modeling language is instead used for easier constraints that developers feel very comfortable with. One possible motivation may be exactly the lack of good tools for consistency checking. Since the ORM editor works as a code generator, errors in an ORM diagram stay undetected until later, when the database is populated. Going back and fixing the diagram is costly at this stage: it requires re-generating the database schema and the consistency code, re-importing data, and possibly dealing with the re-integration of hand-written code that was implemented against the faulty database. It is, thus, possible that tools like ours will encourage the use of more complex constraints.

6. RELATED WORK

There are several earlier approaches to automatic example data generation. Compared to other work on checking the satisfiability of ORM, ours is distinguished by its completeness (for ORM$^-$) and emphasis on practicality. Jarrar and Heymans [14] use the fact that "no complete satisfiability checker is known for ORM" to motivate the introduction of 9 *un*satisfiability patterns that capture conceptual modeling mistakes. The presence of these patterns can be checked with a program search. Heymans [12] describes a translation of a subset of ORM to a formalism with an ExpTime decision procedure. Keet [15] reduces another subset of ORM (emphasizing ring constraints) to Description Logic languages. These reductions follow the general pattern of the standard translation of ORM to logic, and are not intended for efficient execution. We are working on the problem from the opposite end, and it may be interesting to try to extend ORM$^-$ when greater expressiveness is needed, rather than to try to restrict larger fragments.

Wilmore and Embury [25] propose an *intensional* approach to database testing, which is closely related in spirit to our test of constraint satisfiability. Nevertheless, they do not address the fundamental undecidability of the language they support, so their technique is heuristic, without a clear understanding of its expressiveness and limitations.

Other database testing techniques, such as that of Deng et al. [3], concentrate on producing test databases with arbitrary values or after pruning illegal values through heuristics. A more formal approach is followed by Neufeld et al. [19]: constraints are translated into logic and a generator specification is derived semi-automatically (user control may be needed) from the logical formula. This approach is, again, heuristic, in that it may not be able to produce appropriate data. Nevertheless, it offers a way to handle complex constraints in a unifying framework.

There is significant work in databases on the satisfiability of different kinds of constraints. Fan and Libkin [6] address XML document specifications with DTDs and integrity constraints. Calvanese and Lenzerini study the interaction between subtype and "cardinality" constraints [2]. (The latter correspond roughly to what we called "frequency" constraints in ORM.) Both of the above pieces of work are interesting because they use integer constraints for some of their formulations. Nevertheless, the kinds of constraints supported are significantly different from the ORM constraints we tackled. For instance, a key element in the Calvanese and Lenzerini work is that a subtype can refine the cardinality constraint of its supertype. E.g., we can specify that objects of type Person appear at least 3 times in some table, yet objects of subtype Man can appear at most 5 times.

In terms of consistency reasoning for modeling languages, UML is a common focus. UML is really a collection of specification languages that cover many aspects of object-oriented development—from problem specification over class diagrams, to module interactions, to software deployment. Egyed presents a fast technique for maintaining the consistency of different UML diagrams [4]. Recent work also proposes actions for fixing inconsistencies [18, 5]. Nevertheless, neither the core problem we target (inconsistency under *all* inputs) nor the kinds of constraints we support are closely related to that work.

Alloy has been applied successfully in several other domains. Warren et al. compile software architecture specifications to Alloy to check their consistency before performing a proposed software architecture reconfiguration [24]. Khurshid and Jackson discovered bugs in the Intentional Naming System (INS) by modeling and checking it with Alloy [16]. Finally, several translation schemes from Java to Alloy have been proposed: Taghdiri's [22] is a good example.

Acknowledgments. Wes Hunter, David Zook, and Molham Aref of LogicBlox gave valuable help on ORM constraint selection and ORM editor usage, as well as helpful comments on this paper. Pete Manolios, Gayatri Subramanian, and Daron Vroon encouraged us to use CoBaSa and helped us with it. This work was funded by NSF:CCR-0735267 and by LogicBlox Inc.

7. REFERENCES

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. PBS: A backtrack search pseudo-boolean solver and optimizer. In *Proc. Int. Symp. on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[2] D. Calvanese and M. Lenzerini. On the interaction between isa and cardinality constraints. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 1994.

[3] Y. Deng, P. G. Frankl, and D. Chays. Testing database transactions with AGENDA. In *Proc. Int. Conf. on Software Engineering (ICSE)*, 2005.

[4] A. Egyed. Instant consistency checking for the UML. In *Proc. Int. Conf. on Software Engineering (ICSE)*, 2006.

[5] A. Egyed. Fixing inconsistencies in UML design models. In *Proc. Int. Conf. on Software Engineering (ICSE)*, 2007.

[6] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *J. ACM*, 49(3):368–406, 2002.

[7] J. Grant and J. Minker. Inferences for numerical dependencies. *Theor. Comput. Sci.*, 41(2-3):271–287, 1985.

[8] T. A. Halpin. A fact-oriented approach to schema transformation. In *Proc. Mathematical Fundamentals of Database and Knowledge Base Systems (MFDBS)*, 1991.

[9] T. A. Halpin. Object-role modeling (ORM/NIAM). In *Handbook on Architectures of Information Systems*, 1998.

[10] T. A. Halpin. *Information Modeling and Relational Databases.* Morgan Kaufmann, 2001.

[11] T. A. Halpin and H. A. Proper. Database schema transformation and optimization. In *Proc. Object-Oriented and Entity-Relationship Modelling (OOER)*, 1995.

[12] S. Heynmans. *Decidable Open Answer Set Programming.* PhD thesis, Vrije Universiteit Brussel, Department of Computer Science, 2006.

[13] D. Jackson. *Software abstractions: Logic, language, and analysis.* MIT Press, 2006.

[14] M. Jarrar and S. Heymans. Unsatisfiability reasoning in ORM conceptual schemes. In *Int. Conf. on Semantics of a Networked World (ICSNW)*, 2005.

[15] C. M. Keet. Prospects for and issues with mapping the object-role modeling language into DLRifd. In *Proc. Int. Workshop on Description Logics*, 2007.

[16] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. Automated Software Engineering (ASE)*, 2000.

[17] P. Manolios, G. Subramanian, and D. Vroon. Automating component-based system assembly. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, 2007.

[18] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proc. Int. Conf. on Software Engineering (ICSE)*, 2003.

[19] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data: Restricting the search space by a generator formula. *VLDB Journal*, 2:173–213, 1993.

[20] G. M. Nijssen and T. A. Halpin. *Conceptual schema and relational database design: A fact oriented approach.* Prentice-Hall, 1989.

[21] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proc. Automated Software Engineering (ASE)*, 2003.

[22] M. Taghdiri. Inferring specifications to detect errors in code. In *Proc. Automated Software Engineering (ASE)*, 2004.

[23] P. van Bommel, A. ter Hofstede, and T. van der Weide. Semantics and verification of object-role models. *Information Systems*, 16(5):471–495, 1991.

[24] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *Proc. Automated Software Engineering (ASE)*, 2006.

[25] D. Willmor and S. M. Embury. An intensional approach to the specification of test cases for database applications. In *Proc. Int. Conf. on Software Engineering (ICSE)*, 2006.