Unbounded Data Model Verification Using SMT Solvers

Jaideep Nijjar and Tevfik Bultan
University of California, Santa Barbara
{jaideepnijjar, bultan}@cs.ucsb.edu

ABSTRACT

The growing influence of web applications in every aspect of society makes their dependability an immense concern. A fundamental building block of web applications that use the Model-View-Controller (MVC) pattern is the data model, which specifies the object classes and the relations among them. We present an approach for unbounded, automated verification of data models that 1) extracts a formal data model from an Object Relational Mapping, 2) converts verification queries about the data model to queries about the satisfiability of formulas in the theory of uninterpreted functions, and 3) uses a Satisfiability Modulo Theories (SMT) solver to check the satisfiability of the resulting formulas. We implemented this approach and applied it to five open-source Rails applications. Our results demonstrate that the proposed approach is feasible, and is more efficient than SAT-based bounded verification.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Data abstraction*

General Terms

Verification

Keywords

Unbounded verification, MVC frameworks, SMT solvers

1. INTRODUCTION

The web has evolved into an ubiquitous medium for computing and communication services that both businesses and individuals rely on extensively. There is reason to be concerned about this ever-increasing reliance on web applications: web application development is an error-prone process that produces a complicated distributed software system with complex interactions among many components. Moreover, due to the extensive use of scripting languages in web application development, static analysis of web software is very difficult.

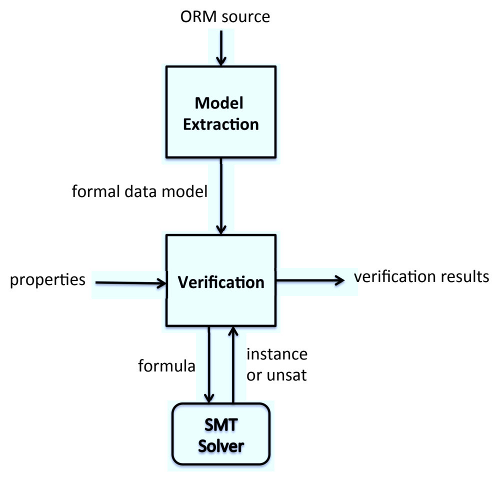

Figure 1: Tool Architecture

One positive advancement in web application development has been the adoption of the Model-View-Controller (MVC) pattern [14]. Many popular web application development frameworks such as Ruby on Rails (Rails for short), Zend for PHP, CakePHP, Django for Python, and Spring for J2EE are based on the MVC pattern. This pattern facilitates the separation of the data model (Model) from the user interface logic (View) and the control flow logic (Controller). The modularity and separation of concerns principles imposed by the MVC pattern provide opportunities for developing customized verification and analysis techniques.

Our work focuses on the verification of data models in web applications. A data model specifies the types of objects (*e.g.*, user, photo, profile, etc.) and the relations among the objects (*e.g.*, the relations between users and photos and profiles) in a web application. A data model also specifies constraints on the data model relations (*e.g.*, the relation between users and profiles must be one-to-one).

In this paper we present an unbounded, automated verification technique and tool (Figure 1) for data model verification of MVC-based web applications. MVC-based frameworks use an object-relational mapping (ORM) to map the data representation of the web application to the back-end database. The front-end of our tool automatically extracts a formal data model from the ORM specification of the input web application. Currently, our front-end only supports ORM specifications of Ruby on Rails web applications. The back-end of our tool converts verification queries to formulas in the theory of uninterpreted functions and then uses a Satisfiability Modulo Theories (SMT) solver to determine the satisfiability of the queries. Based on the output of the SMT solver, our tool reports whether the property holds or fails, and for failing assertions it also reports a data model instance as a counter-example. Our SMT-based verification approach does not bound the sizes of the object classes or the relations, so if our verification tool reports that an assertion holds, it is guaranteed to hold for any data model instance.

We applied our approach to five open-source Ruby on Rails applications and identified a variety of data model errors. Our results show that our verification technique is feasible and useful in analyzing real-world applications. We also compared the performance of our approach with a bounded-verification approach based on SAT solvers, and our experiments demonstrate that, in addition to guaranteeing correctness for verified properties (which is not possible with bounded verification), surprisingly, our unbounded verification approach is more efficient than bounded verification when the bound on the class size exceeds 10.

Our contributions in this paper include: 1) An automated translation of data model verification queries to formulas in the theory of uninterpreted functions. 2) A new data model verification technique for MVC-based web applications that uses SMT solvers and can handle unbounded data models. 3) A technique for reducing the size of the data model specification by projecting it based on the property that is being verified. 4) Experimental evaluation of the proposed approach on five real-world applications.

The rest of the paper is organized as follows: Section 2 discusses data models in web applications and gives a formalization of the data model verification problem. Section 3 presents the translation of data model verification queries to the theory of uninterpreted functions. Section 4 explains our data model projection technique. Section 5 presents our experimental results. Related work is discussed in Section 6 and Section 7 concludes the paper.

2. WEB APPLICATION DATA MODELS

The data model forms the foundation of an MVC-based web application. Any error in this foundation can have a significant impact on the entire application. The data model provides an abstraction between the application code and the backend datastore in a web application. Typically, from the backend datastore's point of view the data is stored in a relational database, whereas from the perspective of the application code the data is represented using an object-oriented data model. The ORM provided by the MVC-based web application development frameworks handles the translation between these two views, so that the application code and the backend datastore can interact with each other while preserving their own views of the data.

```
1   class User < ActiveRecord::Base
2           has_and_belongs_to_many :roles
3           has_one :profile, :dependent => :destroy
4           has_many :photos, :through => :profile
5   end
6   class Role < ActiveRecord::Base
7           has_and_belongs_to_many :users
8   end
9   class Profile < ActiveRecord::Base
10          belongs_to :user
11          has_many :photos, :dependent => :destroy
12          has_many :videos, :dependent => :destroy,
13                           :conditions => "format='mp4'"
14  end
15  class Photo < ActiveRecord::Base
16          belongs_to :profile
17          has_many :tags, :as => :taggable
18  end
19  class Video < ActiveRecord::Base
21          belongs_to :profile
22          has_many :tags, :as => :taggable
23  end
24  class Tag < ActiveRecord::Base
25          belongs_to :taggable, :polymorphic => true
26  end
```

Figure 2: A data model example

The ORM used in the Ruby on Rails framework is called Active Records. Figure 2 shows a simple Active Records specification for a social networking application where users have profiles which store their photo and video files. The photos and videos can be tagged by users, and users can have different roles. Below, we explain different types of declarations supported by Active Records using this example, and we discuss how these declarations can be formalized as relational constraints on a formal data model.

We define a data model as a tuple $M = \langle S, C, D \rangle$ where S is the data model schema identifying the sets and relations of the data model, C is a set of relational constraints, and D is a set of dependency constraints. The schema S identifies the names of the object classes and the names and domains of the relations in the data model. For example, the schema for the example shown in Figure 2 will identify the following set of object classes {User, Role, Profile, Photo, Video, Tag} and the relations among these object classes {photo-profile, photo-tag, photo-user, profile-user, profile-video, role-user, tag-video}, The relational constraints in C express the constraints on these relations that are imposed by their declarations. If a given relation r satisfies a given constraint, then we would state that $r \models C$.

Basic Relation Declarations. Rails supports three basic types of relations among objects: 1) *one-to-one*, 2) *one-to-many*, and 3) *many-to-many*. The `has_one` and `belongs_to` declarations in lines 3 and 10 in Figure 2 define a one-to-one relation between the User and Profile classes. More accurately, this is a one-to-zero-or-one relation and it declares that each User object must be associated with zero or one Profile object, and each Profile object must be associated with exactly one User object. In order to formalize this relation as a constraint, let us use o_U and o_P to denote the set of objects for the User and Profile classes and r_{U-P} to denote the relation between User objects and Profile objects.

Then the constraint that corresponds to this relation is formalized as:

$$\begin{aligned} & (\forall p \in o_P, \exists u \in o_U, (u,p) \in r_{U-P}) \\ \wedge \quad & (\forall p, p' \in o_p, \forall u \in o_U, \\ & ((u,p) \in r_{U-P} \wedge (u,p') \in r_{U-P}) \Rightarrow p = p') \\ \wedge \quad & (\forall p \in o_p, \forall u, u' \in o_U, \\ & ((u,p) \in r_{U-P} \wedge (u',p) \in r_{U-P}) \Rightarrow u = u') \end{aligned} \quad (1)$$

Next, let us consider the one-to-many relation between the Profile and Photo classes, which is declared in the Rails data model in Figure 2 using the `has_many` and `belongs_to` declarations in lines 11 and 16. Using o_P and o_{Ph} to denote the set of objects for the Profile and Photo classes and r_{P-Ph} to denote the profile-photo relation, the formal data model constraint that corresponds to this declaration is:

$$\begin{aligned} & (\forall ph \in o_{Ph}, \exists p \in o_P,\ (p, ph) \in r_{P-Ph}) \\ \wedge \quad & (\forall p, p' \in o_P, \forall ph \in o_{Ph}, \\ & ((p, ph) \in r_{P-Ph} \wedge (p', ph) \in r_{P-Ph}) \Rightarrow p = p') \end{aligned} \quad (2)$$

Finally, a many-to-many relation can be expressed in Rails using the `has_and_belongs_to_many` declaration on both sides of the relation as shown in lines 2 and 7 in Figure 2. This declares a many-to-many relation between the User and Role classes. For such declarations we do not have to create any additional constraints since any relation is a many-to-many relation.

Extensions. Rails provides a set of options that can be used to extend the three basic relations mentioned above. The first option is the `:through` option for the `has_many` and `has_one` declarations. The `:through` option enables the declaration of new relations that are the composition of two other relations. Consider line 4 in Figure 2 which ends with `:through => :profile` and declares a relation between User and Photo objects. When this declaration is combined with the declarations of the relation between User and Profile objects (lines 3 and 10) and Profile and Photo objects (lines 11 and 16), it specifies that the relation between the User and Photo objects (r_{U-Ph}) is the composition of the relations between the User and Profile objects (r_{U-P}) and the Profile and the Photo objects (r_{P-Ph}). This can be formalized as:

$$\begin{aligned} & \forall u \in o_U, \forall ph \in o_{Ph},\ (u, ph) \in r_{U-Ph} \Leftrightarrow \\ & (\exists p \in o_P,\ (u,p) \in r_{U-P} \wedge (p, ph) \in r_{P-Ph}) \end{aligned} \quad (3)$$

The second option that can be used to extend relations is the `:conditions` option, which can be set on all of the four declarations (`has_one`, `has_many`, `belongs_to`, and `has_and_belongs_to_many`). The `:conditions` option limits the relation to those objects that meet a certain criteria. For example, based on the relation declaration in lines 12 and 13 in Figure 2, `Video` objects are only related to a `Profile` object if their `format` field is `mp4`. (Note that the condition statement needs to be in the form of the WHERE clause of a SQL query.) The formalization of this constraint defines a set of objects ($o_{V'}$) that is a subset of the Video objects (o_V) (corresponding to Video objects with format field "mp4") and restricts the relation between the Profile and Video objects (r_{P-V}) to that subset. Formally:

$$o_{V'} \subseteq o_V \wedge (\forall p \in o_P, \forall v \in o_V, (p, v) \in r_{P-V} \Rightarrow v \in o_{V'}) \quad (4)$$

Rails also supports the declaration of polymorphic associations. This is similar to the idea of interfaces in object oriented design, where dissimilar things may have common characteristics that are embodied in the interface they implement. In Rails, polymorphic associations are declared by setting the `:polymorphic` option on the `belongs_to` declaration and the `:as` option on the `has_one` or `has_many` declarations. We see the use of the `:polymorphic` option in Figure 2 between Tags, Photos and Videos (lines 17, 22, 25). Photos and Videos do not have a sub-class relationship but they both can have Tags. The use of the `:polymorphic` option in the Tag class creates a relationship which allows any class to act as a Taggable object and relate to the Tag class via this relation. This is formalized by defining a set of objects for the superset and then expressing inheritance using subset constraints. For the example above, we define a new set of objects called Taggable (o_T), the superset, and declare that Video objects (o_V) and Photo objects (o_{Ph}) are mutually exclusive subsets of the Taggable objects as follows:

$$o_V \subseteq o_T \wedge o_{Ph} \subseteq o_T \wedge o_V \cap o_{Ph} = \emptyset \quad (5)$$

Now a relation can be formally specified between Tag and Taggable objects using ideas discussed earlier.

Dependency Constraints. The final Rails construct we want to discuss adds some dynamism to the data model. It allows the modeling of object deletion at the data model level. The Rails construct for this is the `:dependent` option, which can be set for all the relation declarations except `:has_and_belongs_to_many`. Normally when an object is deleted, its related objects are not deleted. However, by setting the `:dependent` option to `:destroy` or `:delete` (`:delete_all` for `has_many`), deleting an object will also delete the associated objects. Although there are several differences between `:destroy` and `:delete`, the one that is important for our purposes is that `:delete` will directly delete the associated objects from the database without looking at their dependencies, whereas `:destroy` first checks whether the associated objects itself has associations with the `:dependent` option set.

In Figure 2 we see that the User class has the `:dependent` option set for the relation with the Profile class (line 3). Thus, when a User object is deleted, the Profile object that is associated with that User will also be deleted. Further, since the `:dependent` option is set to `:destroy`, any relations of the Profile class with the `:dependent` option set will cause those associated objects to be deleted as well.

Formal modeling of the dependency constraints requires us to model the delete operation. Consider the relation between the User and Profile objects. In order to model the delete operation we have to specify the set of User objects, the set of Profile objects and the relation between the User and Profile objects both before and after the delete operation (o_U, o'_U, o_P, o'_P, r_{U-P}, and r'_{U-P}, respectively). Then we need to specify that when a User object is deleted, the Profile objects related to that User are also deleted. Formally:

$$\begin{aligned} & o'_P \subseteq o_P \wedge o'_U \subseteq o_U \wedge r'_{U-P} \subseteq r_{U-P} \\ \wedge \quad & (\exists u \in o_U, u \notin o'_U \wedge (\forall u' \in o_U, u' \neq u \Rightarrow u' \in o'_U) \\ \wedge \quad & (\forall p \in o_P, (u,p) \in r_{U-P} \Rightarrow p \notin o'_P) \\ \wedge \quad & (\forall p \in o_P, (u,p) \notin r_{U-P} \Rightarrow p \in o'_P) \\ \wedge \quad & (\forall u' \in o_U, \forall p \in o_P, ((u',p) \in r_{U-P} \wedge (u,p) \notin r_{U-P}) \\ & \Rightarrow (u',p) \in r'_{U-P})) \end{aligned} \qquad (6)$$

The constructs we have discussed above form the essence of Rails data models. Similar constructs are also supported by other ORMs such as CakePHP, which has the equivalent of the four basic Rails association declarations and all declaration options except for the `:polymorphic`, and Django, which has the ability to create all three basic relations and `:through` relations like in Rails, but none of the remaining features. Using such constructs, a developer can specify complex relations among objects of an application. Since a typical application would contain dozens of object classes with many relations among them, it is possible to have errors and omissions in the data model specification that can result in unexpected behaviors and bugs. Hence, it would be worthwhile to automatically verify the data models. Below, we formalize the data model verification problem.

Formalizing Verification Queries. In order to formalize verification queries, we first define data model instances and what it means for the data model instance to satisfy a given set of data model constraints.

A data model instance is a tuple $I = \langle O, R \rangle$ where $O = \{o_1, o_2, \ldots o_{n_O}\}$ is a set of object classes and $R = \{r_1, r_2, \ldots r_{n_R}\}$ is a set of object relations and for each $r_i \in R$ there exists $o_j, o_k \in O$ such that $r_i \subseteq o_j \times o_k$.

Given a data model instance $I = \langle O, R \rangle$, we write $R \models C$ to denote that the relations in R satisfy the constraints in C. Similarly, given two instances $I = \langle O, R \rangle$ and $I' = \langle O', R' \rangle$ we write $(R, R') \models D$ to denote that the relations in R and R' satisfy the constraints in D.

A data model instance $I = \langle O, R \rangle$ is an *instance* of the data model $M = \langle S, C, D \rangle$, denoted by $I \models M$, if and only if 1) the sets in O and the relations in R follow the schema S, and 2) $R \models C$.

Given a pair of data model instances $I = \langle O, R \rangle$ and $I' = \langle O', R' \rangle$, (I, I') is a *behavior* of the data model $M = \langle S, C, D \rangle$, denoted by $(I, I') \models M$ if and only if 1) O and R and O' and R' follow the schema S, 2) $R \models C$ and $R' \models C$, and 3) $(R, R') \models D$.

Given a data model $M = \langle S, C, D \rangle$, we will define four types of properties: 1) *state assertions* (denoted by A_S): these are properties that we expect to hold for each instance of the data model; 2) *behavior assertions* (denoted by A_B): these are properties that we expect to hold for each pair of instances that form a behavior of the data model; 3) *state predicates* (denoted by P_S): these are predicates we expect to hold in some instance of the data model; and, finally, 4) *behavior predicates* (denoted by P_B): these are predicates we expect to hold in some pair of instances that form a behavior of the data model. We will denote that a data model satisfies an assertion or a predicate as $M \models A$ or $M \models P$, respectively. Then, we can use the following formal definitions for these four types of properties:

$$\begin{aligned} M \models A_S \quad &\Leftrightarrow \quad \forall I = \langle O, R \rangle, I \models M \Rightarrow R \models A_S \\ M \models A_B \quad &\Leftrightarrow \quad \forall I = \langle O, R \rangle, \forall I' = \langle O', R' \rangle \\ & \qquad (I, I') \models M \Rightarrow (R, R') \models A_B \\ M \models P_S \quad &\Leftrightarrow \quad \exists I = \langle O, R \rangle, I \models M \wedge R \models P_S \\ M \models P_B \quad &\Leftrightarrow \quad \exists I = \langle O, R \rangle, \exists I' = \langle O', R' \rangle), \\ & \qquad (I, I') \models M \wedge (R, R') \models P_B \end{aligned}$$

To perform verification of data models, we use the formalization presented above to convert verification queries about the data model to satisfiability of formulas in the theory of uninterpreted functions. We then use an SMT solver to answer the verification queries, as we discuss next.

3. TRANSLATION TO SMT

SMT-LIB is the standard input language for SMT solvers [18]. We have implemented a translator that takes Rails Active Record files describing the data model as input and generates an SMT-LIB specification for the data model. The generated SMT-LIB specification consists of a conjunction of constraints in the theory of uninterpreted functions. In this section we describe how different Active Record constructs that define the data model can be translated to constraints in the theory of uninterpreted functions.

SMT-LIB specifications are written as sequences of s-expressions. Uninterpreted functions are declared using the `declare-fun` command and types are declared using the `declare-sort` command. For example, the specification

```
(declare-sort Video 0)
(declare-fun isMp4Video (Video) Bool)
```

declares a `Video` type (that takes 0 parameters) and a boolean function called `isMp4Video` that accepts a value of type `Video`.

SMT-LIB supports the basic boolean operators (`not, and, or`), equality (`=`), implication (`=>`), and if-then-else (`ite`). Quantifiers are expressed using the `forall` and `exists` operators. Constraints are specified using the keyword `assert`.

After this short overview of the SMT-LIB language syntax, we now explain how we translate the formal model constraints discussed in Section 2 to SMT-LIB. Let us first consider constraint (3) from Section 2 which characterizes the semantics of a one-to-many relation declaration. We translate the one-to-many relation between the Profile and Photo classes to SMT-LIB using an uninterpreted function as:

```
(declare-sort Profile 0)
(declare-sort Photo 0)
(declare-fun profile_photo (Photo) Profile)
```

where a Photo and a Profile object are related if and only if the `profile_photo` function maps one to the other.

Constraint (1) in Section 2 represents the semantics of a one-to-one relation declaration. We translate such a relation to SMT-LIB using an uninterpreted function above, but adding an extra constraint restricting the cardinality of the relation. For example, the one-to-one relation between User and Profile is translated as:

```
(declare-sort User 0)
(declare-sort Profile 0)
(declare-fun user_profile (Profile) User)
(assert (forall ((p1 Profile)(p2 Profile))
   (=>  (not  (= p1 p2))
     (not (= (user_profile p1) (user_profile p2) ))
) ))
```

Note that the above constraint specifies each User is associated with one or no Profile and each Profile is associated with exactly one User as we expect based on the semantics of the one-to-one relation declaration.

Since uninterpreted functions map each input value to a single value in the range, it is not possible to represent a many-to-many relation between two sets using an uninterpreted function with one parameter as we did for the one-to-one and one-to-many relations. Instead, we translate a many-to-many relation declaration to SMT-LIB by declaring an uninterpreted boolean function with two arguments that returns true if and only if the two objects passed in as arguments are related. For instance, a many-to-many relation between the User and Role classes is translated as:

```
(declare-sort User 0)
(declare-sort Role 0)
(declare-fun user_role (User Role) Bool)
```

As discussed in Section 2, relations that are the composition of other relations can be declared in a data model using the `:through` keyword and constraint (4) provides a formalization of such declarations. For example, assume that Users are associated with Profiles, Profiles are associated with Photos and the data model declares a third relation between Users and Photos such that it is the composition of the other two relations. This is translated to SMT-LIB as[1]:

```
(declare-sort Profile 0)
(declare-sort Photo 0)
(declare-sort User 0)
(declare-fun profile_photo (Photo) Profile)
(declare-fun user_profile (Profile) User)
(declare-fun user_photo (Photo) User)
(assert (forall ((u User)(ph Photo))
    (iff  (= u (user_photo ph))  (exists ((p Profile))
    (and (= u (user_profile p)) (= p (profile_photo ph))  ))
) ))
```

The `:conditions` option is used to express a relationship between one set of objects and the subset of another set of objects and is formalized in constraint (5) of Section 2. Since there is no support for subtyping or inheritance in the SMT-LIB language, we model the `:conditions` option by creating a boolean function that returns true if and only if the argument object is in the designated subset. To give a concrete example, say Videos are associated with a Profile only if the Profile is active. The SMT-LIB translation of such a declaration would be:

```
(declare-sort Video 0)
(declare-sort Profile 0)
(declare-fun isActive (Profile) Bool)
(declare-fun activeprofile_video (Profile) Video)
(assert (forall ((p Profile)(v Video))
  (=> (= v (activeprofile_video p)) (isActive p) )
))
```

[1]The if and only if operator, `iff`, is used here for clarity. This can easily be converted into a double implication to conform to the official SMT-LIB set of operators.

Here, the `isActive` function is used to characterize the subset of Profiles that are active, and the final constraint ensures that the function `activeprofile_video` only returns active Profiles.

Next, Rails Active Records support the specification of polymorphic relations as discussed in Section 2 and formalized in constraint (6). In order for one type to be related to multiple other types, one can create a supertype that the former type can relate to. For example, if a Tag can be related to both Photos and Video, we can create a supertype of Photo and Video that Tag can be related to. Let us call this supertype Taggable and let Photo and Video be subtypes of it. As mentioned earlier, SMT-LIB does not support subtyping so we use boolean functions to model such a declaration. We also add a constraint that states the Taggable type is abstract, i.e. all Taggable objects are either Photos or Videos, and that these subtypes are mutually exclusive:

```
(declare-sort Tag 0)
(declare-sort Taggable 0)
(declare-fun isPhoto (Taggable) Bool)
(declare-fun isVideo (Taggable) Bool)
(assert (forall ((t Taggable)) (and
    (or (isPhoto t) (isVideo t) )
    (iff (isPhoto t) (not (isVideo t)) )
)))
(declare-fun taggable_tag (Tag) Taggable)
```

This example shows a simple case of polymorphic relations. In general, a class may be polymorphically-related to multiple classes. For instance, Multimedia may have a polymorphic relation with the Video and Audio classes. Combined with the scenario above, Video will now require two supertypes (say Taggable and MultimediaItem). In our tool we actually create one ultimate supertype called PolymorphicClass of which any polymorphically-related types are subtypes (such as Photo, Video, and Audio) as well as their supertypes (Taggable and MultimediaItem). All these subtypes are expressed in SMT-LIB language using boolean functions. Then an assert is added which contains constraints specifying which types are subtypes of which supertypes, that subtypes are mutually exclusive of others in the same supertype, that the supertypes themselves are abstract (meaning all elements belong to one of its subtypes), and that PolymorphicClass is also abstract. Furthermore, since subtypes (such as Photo and Video) are not types of their own (i.e. no sort is declared for them), any non-polymorphic relations with these classes require an assert that enforces the range of the function, similar to constraint (5).

Finally, let us discuss delete dependencies that are declared using the `:dependent` option and formalized with constraint (7) in Section 2. This type of constraint expresses a change from one state of the model (before an object is deleted) to another (the state of the model after the object deleted, i.e., post-delete state). We model the post-delete state in SMT-LIB translation using boolean functions (denoted with the prefix "Post"). There is one such function for every type. This function returns true if the object exists after the delete operation. For example:

```
(declare-sort User 0)
(declare-fun Post_User (User) Bool)
```

There is also one such boolean function for every relation. This function returns true if and only if the two objects are still related after the deletion occurs. For instance:

```
(declare-fun user_profile (Profile) User)
(declare-fun Post_user_profile (Profile User) Bool)
```

When one wants to perform a verification query about how the deletion of an object affects other objects and relations, these boolean functions are used to express the property. For example, to express a property about deleting a User, our translator generates a constraint that defines the Post_object functions as well as the Post_relation functions according to the dependencies expressed in the data model. The algorithm to generate this takes into account the effect of dependencies on transitive, conditional and polymorphic relations. Our algorithm assumes no cyclic delete dependencies. Encoding cyclic dependencies requires transitive closure, which is not expressible in the theory of uninterpreted functions. Here is the constraint generated by our translation algorithm for the simple User-Profile scenario of deleting a User, where `x` denotes the User being deleted:

```
(assert (not (forall ((x User))  (=> (and
  (forall ((a User)) (ite (= a x)
     (not (Post_User a)) (Post_User a)))
  (forall ((b Profile)) (ite (= x (user_profile b))
     (not (Post_Profile b)) (Post_Profile b)  ))
  (forall ((a Profile) (b User)) (ite
     (and (= b (user_profile a)) (Post_Profile a))
       (Post_user_profile a b)
       (not (Post_user_profile a b)) ))
  ) ;Remaining property-specific constraints go here
)))
```

4. DATA MODEL PROJECTION

Our tool checks the correctness of each verification query separately, and this creates an opportunity for reducing the size of the generated SMT-LIB specifications. Reducing the size of the generated SMT-LIB specifications reduces the cost of the satisfiability check and hence increases the performance of our tool. The basic idea is the following: Given a property to verify, we can reduce the size of the generated SMT-LIB specification by removing the declarations and constraints about the parts of the data model that does not depend on the property that we are planning to verify. We call this technique *property-based data model projection*.

We formally define the property-based data model projection as a function, denoted by Π, that takes a data model and a property as input and returns a new data model. Hence, given a data model $M = \langle S, C, D \rangle$ and a property p, $\Pi(M, p) = M_p$ where $M_p = \langle S, C_p, D_p \rangle$ is the projected data model such that $C_p \subseteq C$ and $D_p \subseteq D$. Note that the projection function removes some of the relational and dependency constraints from the model, therefore reducing the size of the projected model. If the property p is a state assertion or a state predicate (denoted by A_S and P_S in Section 2), then the projection function Π removes all the dependency constraints (i.e., $D_p = \emptyset$) since dependency constraints are only relevant for behavior assertions and predicates (denoted by A_B and P_B in Section 2).

A key property of the projection function Π is that it preserves the correctness of the input property. Formally, $M \models p \Leftrightarrow \Pi(M, p) \models p$, for any property p.

Algorithm 1 Data Model Projection Algorithm

```
Input: model: Rails Active Records files; p: property; delclass: the
  class name for the deleted object (only needed when p is a behavior
  assertion or predicate)
Output: Projected SMT-LIB specification

  pclasses := list of classes mentioned in p
  prelations := list of relations mentioned in p
  if p is a behavior assertion or predicate then
     Follow dependencies for delclass with respect to the relations
     given in prelations
     Add any dependent classes, and the relations through which
     they are dependent, to pclasses and prelations
  end if
  for all class in pclasses do
     Output an uninterpreted function declaration and Post function
     declaration for class
     if there exists a relation in prelations that has a conditional
     relation with class then
        Output a boolean function declaration that models the conditional subset
     end if
  end for
  Output the polymorphic constraints for the polymorphic classes in
  pclasses
  for all relation in prelations do
     Output a function declaration, any associated constraints based
     on the declaration of the relation, and the Post function declaration
     for the relation
     if relation is a transitive relation that is composition of multiple
     relations then
        Output function declarations, associated constraints, and
        Post function declarations for all relations that are part of the composition
     end if
  end for
```

Let us now explain why this property holds. In our verification approach, all verification queries are translated to satisfiability queries. Hence, the above property is equivalent to stating that the SMT-LIB specification we generate from the original model M and the property p is satisfiable if and only if the SMT-LIB specification we generate from the projected model $\Pi(M, p)$ and the property p is satisfiable. Note that, if the SMT-LIB specification we generate from the model $\Pi(M, p)$ and the property p is not satisfiable, then the SMT-LIB specification we generate from the original model M and the property p cannot be satisfiable since the projection operation $\Pi(M, p)$ only removes constraints, resulting in a less constrained SMT-LIB specification. However, our projection algorithm also guarantees that if the SMT-LIB specification we generate from the model $\Pi(M, p)$ and the property p is satisfiable, then the SMT-LIB specification we generate from the original model M and the property p is also satisfiable. This is true due to two reasons: 1) The constraints that the projection function deletes from the original model can never be self-contradictory since they correspond to class and relation declarations, and it is not possible to declare a self-contradictory data model that does not allow any instances using the constructs we analyze; 2) The constraints that the projection function deletes from the original model cannot contradict with the verified property p since the projection algorithm only deletes a class or a relation if that class or relation has no influence on the property p.

We implemented this property-based data model projection as part of our verification tool. The algorithm used is given in Algorithm 1. It requires as input the Rails data model and the property the user wishes to verify about the data model. If the property is a behavior predicate or assertion, it also requires the class name for the object to be deleted. The projected SMT-LIB specification output by the translator contains constraints on only those classes and relations that are explicitly mentioned in the property and the classes and relations that are related to them based on transitive relations, dependency constraints or polymorphic relations.

5. EXPERIMENTS

We used five open-source Ruby on Rails web applications for evaluating the effectiveness of our SMT-based unbounded data model verification approach. We wrote ten properties about each application's data model that we expected to hold based on the semantics of the application. Then using our tool, we generated an SMT-LIB specification for each application. This specification is conjoined with a property and then sent to the SMT solver for satisfiability check.

We used Microsoft's SMT solver, Z3 [20], in our experiments. In addition to returning unsatisfiable or satisfiable, an SMT solver may also return "unknown" or it may timeout since the quantified theory of uninterpreted functions is known to be undecidable [4]. In our experiments the timeout limit was set to five minutes.

Assertions that are verified using our approach are guaranteed to hold in the application. However, assertions that fail may or may not hold in the application—the verification results simply indicate that the property was not enforced by the application's data model. It may not be possible to observe the failure during program execution since the property may actually be enforced in parts of the application that we do not model (*e.g.*, in the Controller code). However, we consider a failed property a data model error if the property could have been enforced statically in the data model but was not. On the other hand, if a failed property cannot be enforced in the data model using the Ruby on Rails constructs, then we do not consider it a data model error. Thus for properties that failed we performed further manual investigation to identify which failing properties were indeed data model errors.

The Applications. Table 1 lists the sizes of the five applications in terms of lines of code, the number of classes, and the number of data model classes. OpenSourceRails (OSR) (http://www.opensourcerails.com) is a social project gallery application that allows users to submit projects, as well as bookmark and rate them. Tracks (http://getontracks.org/) is an application that lets users create and manage to-do lists, where lists can be organized by context and project. FatFreeCRM (http://www.fatfreecrm.com/) is a customer relationship management software that organizes a business' customers, campaigns, opportunities, and accounts. Substruct (http://code.google.com/p/substruct/) is an e-commerce application where users can add products to cart and create wishlists. LovdByLess (http://lovdbyless.com/) is a social networking site with the usual features such as user profile with pictures, becoming friends, etc.

Verification Results. The properties we checked on these applications are listed in Table 2, along with their type from Section 2. (A_S for state assertions, A_B for behavior assertions, P_S for state predicates, and P_B for behavior predicates).

Table 1: Sizes of the Applications

	LOC	Classes	Data Models
LovdByLess	3787	61	13
Tracks	6062	44	13
OSR	4295	41	15
FatFreeCRM	12069	54	20
Substruct	15639	85	17

The results of the verification are also shown in Table 2. ✓indicates that the property was verified and × indicates that the property failed. A total of sixteen properties we tried to verify failed. We investigated each of these failures manually to determine if they correspond to data model errors.

For example, Property L5 from LovdByLess does not hold due to the limited expressiveness in Rails constructs. There is no construct that allows you to add a constraint to a relation expressing that an object cannot be related to itself. Thus the application programmer in LovdByLess had to add a validation function to the model to ensure that a user cannot create a Friend request for herself. Thus the failure of the property L5 does not indicate a data model error.

Another property that failed was property O2. The setup in OpenSourceRails is that a user can bookmark projects, so a User has_many Bookmarks, a Bookmark belongs_to a Project, and a Project has_many Bookmarks. The property O2 states that a User is allowed to Bookmark a Project at most once. However, the declarations used to set up these relationships allow the same user to create multiple bookmarks with the same Project. Thus the application programmer had to enforce this property using the user interface and code in the controller. However, what the programmer desires is a many-to-many relationship between User and Project, as opposed to two one-to-many relationships. One reason for such a setup is if they wanted to hold extra information in the Bookmark class. Investigation into this class shows that this is not the case; hence this corresponds to a data model error in this application.

A failing property indicates the application's data model does not satisfy the property; however, the property may still hold in the overall application because the property is being enforced outside of the data model, as in property O2. However, this was not the case for property O6. This property failed because the declaration in the User model does not have the :dependent option set. Thus a User's associated Bookmarks are not deleted, causing the property to fail. Because the deletion of a user leaves orphaned Bookmarks in the database, property O6 is an example of a data model error that is also an error in the application.

In total we discovered eleven data modeling errors from the sixteen properties that failed. There were two data model errors in LovdByLess, three each in Tracks and OSR, one in Substruct and two in FatFreeCRM. The fact that we were able to discover data model errors in real-world applications is evidence that our approach can be an effective verification approach in practice.

Performance. To further evaluate the effectiveness of our approach, we measured performance of the verification task. Specifically, we measured the verification time reported by Z3 and the number of variables and clauses produced in the SMT specification. By number of variables we mean the number of sorts, functions, and quantified variables. By number of clauses we mean the number of asserts, quantifiers and operations in the SMT-LIB specification.

The measurements were taken for the SMT-LIB specification generated for each property. The values were then averaged over the properties for each application. The results for the verification times are given in Table 3. What we immediately noticed is that the verification is extremely fast; the longest verification time is just 0.025 seconds. One thing to note is that these values do not include the times for those properties that timed out during verification. There were four such properties, all for FatFreeCRM. So although unbounded verification is very quick, the disadvantage is that some properties may not give an answer to the verification query.

The difficulty the SMT solver is having when it times out is due to the number of quantifiers in the SMT specification. To minimize this number we ran the experiments again, this time using the data model projection algorithm discussed in Section 4. With projection we were able to obtain answers to all of the properties that timed out. Further, the verification time decreased for all properties. On average, using the data model projection the verification time decreased by 40%.

To compare effectiveness between bounded and unbounded verification, we used the Alloy Analyzer [13] to verify the same set of properties. The data models of all five applications were run through the ActiveRecord to Alloy translator that was implemented in our previous work [16]. The same measurements were taken as with Z3, but over an increasing bound from at most 10 objects for each class to at most 35 objects for each class. These values were plotted alongside the Z3 verification time for each application where the solver did not time out. As seen in Figure 3, unbounded verification is much faster than bounded verification, even for the smallest bound of 10 objects. Bounded verification using Alloy took up to tens of seconds whereas Z3, took less than a second. However, since Z3 is not guaranteed to terminate because we are generating SMT-LIB specifications in the theory of uninterpreted functions with quantification, we observe that bounded and unbounded verification can be complementary approaches since bounded verification can be used when the unbounded approach fails.

Since Alloy specializes in the verification of object models, it is rather surprising that there is such a drastic difference between the verification times of Z3 and Alloy. There may be several reasons why unbounded verification did so well. Z3 uses many heuristics to eliminate quantifiers in formulas. It uses an E-graph to instantiate quantified variables which, in conjunction with code trees, an inverted path index and eager instantiation, makes it very effective at dealing with quantifiers [8]. Note that these heuristics do not affect the soundness of the verification. Another reason why Z3 performed better than Alloy may be due to their implementation languages: Z3 is implemented in C++ whereas Alloy

Table 2: Verification Results

	LovdByLess Properties	
A_S	L1. A Forum Post is always associated with a Topic	✓
P_S	L2. A Forum Topic may have no Forum Posts	✓
A_S	L3. A Photo is always associated with a user Profile	✓
A_S	L4. Profile's FeedItems = Profile's Feed's FeedItems	✓
A_S	L5. A User can't be her own Friend	×
A_B	L6. Deleting user Profile deletes Photos	×
A_B	L7. Deleting user Profile doesn't delete any Friends	✓
A_B	L8. Deleting a user Profile leaves no orphan Users	×
A_B	L9. Deleting a Message doesn't delete a User	✓
A_B	L10. Deleting a Forum Topic leaves no dangling Posts	✓
	Tracks Properties	
A_S	T1. Every Todo has a Context	✓
P_S	T2. A Context may have no Todos	✓
P_S	T3. Todo can have no associated Project	×
A_S	T4. Note's User = Note's Project's User	×
A_S	T5. Every User has a Preference	×
A_B	T6. Deleting a Project leaves no dangling Notes	✓
A_B	T7. Deleting a Preference leaves no orphan Users	×
A_B	T8. Deleting a User leaves no dangling Contexts	✓
A_B	T9. Deleting a User leaves no dangling Projects	✓
A_B	T10. Deleting a Context leaves no dangling Todos	✓
	OSR Properties	
P_S	O1. A Project can have multiple Screenshots	✓
P_S	O2. A User can Bookmark a Project at most once	×
P_S	O3. A User can Bookmark her own submitted Project	✓
A_S	O4. Project's Bookmark's User = Project's User	✓
P_S	O5. A User can put multiple Comments on one Project	✓
A_B	O6. Deleting a User deletes her Bookmarks	×
A_B	O7. Deleting a User deletes her Activities	×
A_B	O8. Deleting a User doesn't delete her Comments	✓
A_B	O9. Deleting a Project deletes its Ratings	×
A_B	O10. Deleting a Project Rating doesn't delete Project	✓
	Substruct Properties	
A_S	S1. Every Cart is associated with a User	✓
P_S	S2. An Product can be on multiple Wishlists	✓
P_S	S3. A Wishlist can be empty	✓
A_S	S4. A Product is on a User's Wishlist at most once	×
P_S	S5. A User can have multiple Orders	✓
A_B	S6. Deleting a Cart doesn't delete its Products	✓
A_B	S7. Deleting a Product deletes it from all Carts	×
A_B	S8. Deleting a User deletes her Orders	×
A_B	S9. Deleting User doesn't delete Items on its Wishlists	✓
A_B	S10. Deleting a Wishlist doesn't delete its Item	✓
	FatFreeCRM Properties	
A_S	F1. Every Task must have a User	✓
A_S	F2. Every Lead belongs to exactly one User	✓
A_S	F3. AccountOpportunity's Opportunity = AccountOpportunity's Account's Opportunity	✓
P_S	F4. A Contact may have no Tasks	✓
A_S	F5. User's Opportunity = User's Campaigns' Opportunity	×
A_B	F6. Deleting a Lead does not delete Contacts	✓
A_B	F7. Deleting Lead does not delete User	✓
A_B	F8. Deleting an Account deletes associated Tasks	✓
A_B	F9. Deleting a Lead leaves no dangling Contacts	×
A_B	F10. Deleting an Account does not delete Contacts	✓

Table 3: Z3 Verification Times in seconds

	not projected	projected
LovdByLess	0.025	0.015
Tracks	0.023	0.014
OSR	0.016	0.012
Substruct	0.025	0.011
FatFreeCRM	0.022	0.013

(as well as the SAT solver it uses, SAT4J) is implemented in Java. Finally, another likely reason that Z3 is more efficient than Alloy is that SMT solvers operate at a higher level of abstraction than SAT solvers. Thus SMT solvers can use information about the structure and semantics of a formula to make inferences about satisfiability more accurately as well as more efficiently than a SAT-based approach which converts the verification to SAT formulas using a Boolean encoding. In fact, due to increasing size of the Boolean encoding, bounded verification suffers from an exponential increase in verification time with increasing bound.

Besides verification time, we also measured the number of clauses and variables created by Alloy's SAT translation. These measurements were averaged over the properties for each application and plotted over increasing scope, as shown in Figure 5.

The number of clauses and variables were also averaged over properties for the SMT-LIB specifications for each application. In Figure 4 is a plot of the number of clauses and variables in the specifications for each application. Both the projected and non-projected versions of the specifications are shown. We see a tremendous 80% decrease in the number of variables and clauses after performing the data model projection. Although the number of variables and clauses in the SMT specification are not directly comparable to the figures produced by Alloy, we can still observe that the SMT formula size is much smaller than the one used by the SAT solver. We also observe that bounded verification has the disadvantage that the size of the formula used by the SAT solver increases exponentially with respect to bound.

Overall our experimental results indicate that unbounded verification using SMT solvers is more efficient than bounded verification. This leads us to conclude that the approach we presented in this paper is a feasible and efficient approach to data model verification.

6. RELATED WORK

There has been prior work on the verification of data models; these works present bounded verification approaches using the Alloy Analyzer. For example, mapping relational database schemas to Alloy has been studied before [7]. Also, translating ORA-SS specifications (a data modeling language for semi-structured data) to Alloy and using Alloy analyzer to find an instance of the input data model has been investigated [19]. However, unlike our work, these approaches are bounded whereas our technique performs unbounded verification. Also the translation is not automated in these earlier efforts. Finally, Alloy has also been used for discovering bugs in web applications related to browser and business logic interactions [3]. This is a different class of bugs than the data model related bugs we focus on in this paper.

There has been some recent work on unbounded verification of Alloy specifications using SMT solvers [9], but to the best of our knowledge this approach has not been implemented yet. Unbounded verification of Alloy specifications may be more challenging than the data model verification problem that we focus on in this paper since the Alloy language provides powerful constructs such as transitive closure. Such

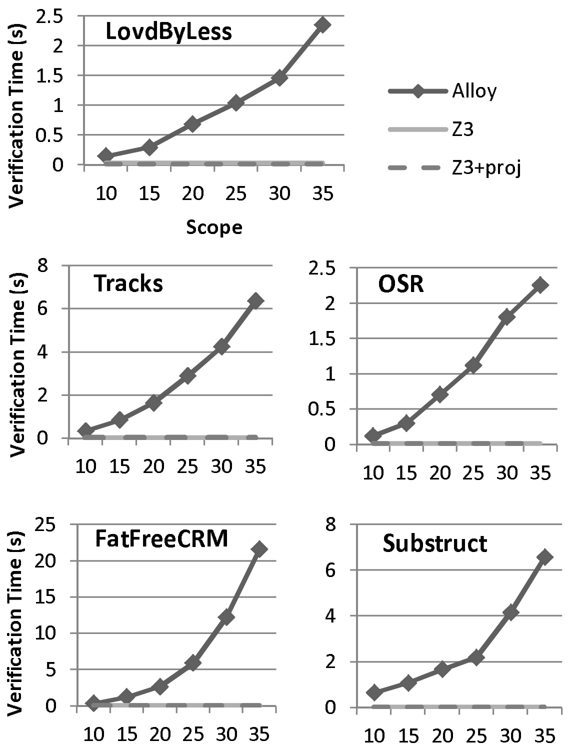

Figure 3: Verification Time, Alloy vs Z3

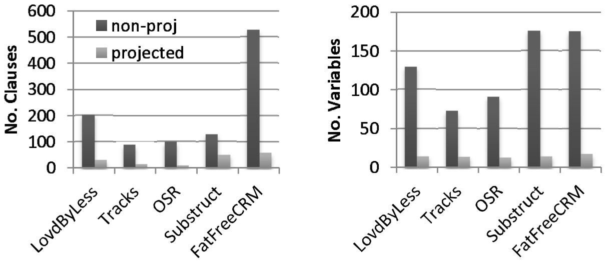

Figure 4: Formula Size, SMT

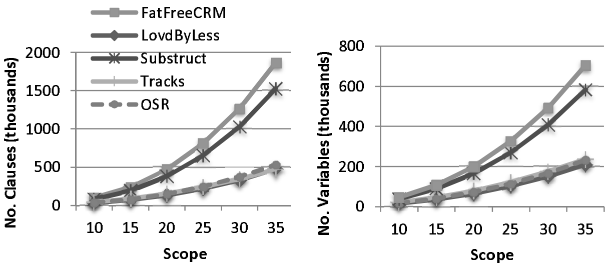

Figure 5: Formula Size, Alloy

constructs do not appear in the data models that we extract from web applications.

There has been recent work on specification and analysis of conceptual data models [17, 15, 11]. These efforts follow the model-driven development approach whereas our approach is a reverse engineering approach that extracts the model of an existing application and analyzes it to find errors.

There are earlier results on the formal modeling of web applications focusing on state machine-based formalisms to capture the navigation behavior (for example, [12, 2, 10]). In contrast to this line of work, we are focusing on analysis of the data model rather than the navigational aspects of the web applications.

There has been earlier work on reducing the cost of automated verification by analyzing dependencies such as cone of influence reduction [1] and program analysis based reductions [5]. Compared to these earlier results, our projection algorithm is a specialized reduction technique for data model verification that utilizes the data model semantics.

The fact that verification with SMT solvers can be more efficient than SAT-based bounded verification has been observed in other verification domains [6]. However, the data model verification problem we investigate in this paper is different from the problems studied in these earlier works.

This paper builds on our earlier work on bounded verification of Rails data models [16]. There are several significant differences in this paper compared to the work presented in [16]: 1) In this paper we use an unbounded verification approach based on SMT-solvers as opposed to the bounded SAT-based verification approach used in [16]. 2) This paper directly constructs formulas in the theory of uninterpreted functions from verification queries about a given Rails data model, whereas the approach presented in [16] translates data models to Alloy specifications which Alloy Analyzer then converts to boolean SAT formulas. 3) We present a novel data model projection technique that simplifies the generated formula by removing constraints about the data model that are not relevant to the property being verified, improving the performance of the analysis. 4) We present experimental analysis demonstrating the performance of the proposed verification approach on five open source Rails applications. 5) We experimentally compare the unbounded verification approach presented in this paper with the bounded verification approach presented in [16] and show that the unbounded verification approach presented in this paper is more efficient.

7. CONCLUSION

We presented an unbounded verification approach for web application data models. We automatically extract a formal data model from the ORM specifications in MVC-based web applications and translate verification queries about these models to satisfiability queries in the theory of uninterpreted functions. We use an SMT solver to check the satisfiability of the resulting formulas. This approach was implemented in a tool that verifies Rails data models. Our experiments demonstrate that our approach is effective in verifying data models of real-world web applications, and that it is more efficient than SAT-based bounded verification.

8. REFERENCES

[1] S. Berezin, S. V. A. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *Proc. COMPOS*, pages 81–102, 1997.

[2] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *Proc. ASE*, pages 100–109, 2004.

[3] B. Bordbar and K. Anastasakis. MDA and analysis of web applications. In *Proc. Work. Trends in Enterprise Appl. Arch.*, pages 44–55, 2005.

[4] R. E. Bryant, S. M. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc. CAV*, pages 470–482, 1999.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Bandera: a source-level interface for model checking java programs. In *Proc. ICSE*, pages 439–448, 2000.

[6] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *Proc. ASE*, pages 137–148, 2009.

[7] A. Cunha and H. Pacheco. Mapping between Alloy specifications and database implementations. In *Proc. SEFM*, pages 285–294, 2009.

[8] L. M. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In *Proc. CADE*, pages 183–198, 2007.

[9] A. A. E. Ghazi and M. Taghdiri. Relational reasoning via SMT solving. In *Proc. FM*, pages 133–148, 2011.

[10] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proc. ASE*, pages 235–244, 2010.

[11] T. Halpin and T. Morgan. *Information Modeling and Relational Databases.* Morgan Kaufmann, 2008.

[12] M. Han and C. Hofmeister. Relating navigation and request routing models in web applications. In *Proc. MoDELS*, pages 346–359, 2007.

[13] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Meth.*, 11(2):256–290, 2002.

[14] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Jour. Object-Orient. Program.*, 1(3):26–49, 1988.

[15] M. J. McGill, L. K. Dillon, and R. E. K. Stirewalt. Scalable analysis of conceptual data models. In *Proc. ISSTA*, pages 56–66, 2011.

[16] J. Nijjar and T. Bultan. Bounded verification of Ruby on Rails data models. In *Proc. ISSTA*, pages 67–77, 2011.

[17] Y. Smaragdakis, C. Csallner, and R. Subramanian. Scalable satisfiability checking and test data generation from modeling diagrams. *Autom. Softw. Eng.*, 16(1):73–99, 2009.

[18] SMT-LIB. http://www.smtlib.org/.

[19] L. Wang, G. Dobbie, J. Sun, and L. Groves. Validating ORA-SS data models using Alloy. In *Proc. ASWEC*, pages 231–242, 2006.

[20] Z3. http://research.microsoft.com/projects/z3/.