An Integrated Data Model Verifier with Property Templates

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

***Abstract*—Most modern web applications are built using development frameworks based on the Model-View-Controller (MVC) pattern. In MVC-based web applications the data model specifies the types of objects used by the application and the relations among them. Since the data model forms the foundation of such applications, its correctness is crucial. In this paper we present a tool, IDAVER, that 1) automatically extracts a formal data model specification from applications implemented using the Ruby on Rails framework, 2) provides templates for specifying data model properties, 3) automatically translates the verification of properties specified using these templates to satisfiability queries in three different logics, and 4) uses automated decision procedures and theorem provers to identify which properties are satisfied by the data model, and 5) reports counterexample instances for the properties that fail. Our tool achieves scalable automated verification by exploiting the modularity in the MVC pattern. IDAVER does not require formal specifications to be written manually; thus, our tool enables automated verification and increases the usability by combining automated data model extraction with template-based property specification.**

I. Introduction

Web applications have become ubiquitous. They are used for commerce, entertainment, social interaction, education and many other tasks. They are globally accessible not only from desktop computers, but from a plethora of mobile devices with network connectivity. Given the significant influence of web applications on modern society, improving their dependability is a critical and crucial problem.

Modern web applications are typically built using development frameworks that are based on the Model-View-Controller (MVC) pattern [6]. MVC-based frameworks include Ruby on Rails (Rails for short), Zend for PHP, CakePHP, Django for Python, and Spring for J2EE. The MVC pattern facilitates the separation of the data model (Model) from the user interface logic (View) and the control flow (Controller).

In this paper, we focus on dependability of data models in web applications. We present IDAVER, a tool for formal verification of data model properties. A data model specifies the types of objects, the relations among the objects and the constraints on the data model relations. MVC-based frameworks use an object-relational mapping (ORM) to map the object-oriented data representation of the web application to the back-end database. By using these ORM specifications to extract a formal data model, we do not require the user to specify a formal data model manually.

Our tool targets web applications developed using the Rails framework. The front-end of our tool automatically extracts a formal data model from the ORM specification of the input application. Although the formal data model is extracted automatically, the user still has to specify the properties she desires to check about the data model. To facilitate this process, we developed a set of property templates. These templates characterize the most common properties we observed in our earlier research on data model verification [10], [11]. These templates can easily be instantiated for different classes and relations by the user.

Our tool verifies properties (specified using property templates) on the automatically extracted formal data model by translating verification queries to satisfiability queries in a specified theory and then using a backend solver for that theory. Our tool combines three different variants of this framework. The first two are SAT-based bounded verification and Satisfiability Modula Theories (SMT)-based unbounded verification from our earlier work [10], [11]. In this paper, we add another unbounded verification approach based on First Order Logic (FOL) and a FOL theorem prover. Our tool integrates these three approaches and provides a unified framework for the verification of data models.

Our contributions in this paper include 1) property templates that facilitate specification of data model properties, 2) a novel data model verification approach that translates verification queries to first order logic (FOL) and uses an automated FOL theorem prover to answer them, and 3) an integrated tool that combines SAT- SMT- and FOL-based data model verification approaches with property templates.

The rest of the paper is organized as follows: Section 2 presents a running example and describes Rails data models. Section 3 defines the formal data model. Section 4 describes the data model property templates. Section 5 presents IDAVER, our tool for integrated data model verification. Section 6 presents a case study. Section 7 discusses the related work and Section 8 concludes the paper.

II. Data Models

In this section we give an overview of Rails' data modeling features using a running example. Figure 1 presents the simplified data model for a social networking application built on the Rails platform. In this application, there are users who create profiles. Photos and videos can be tagged and posted to a user's profile, and users can be attributed with various roles.

```
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
```

Fig. 1: A data model example

A. The Basic Relationships

Rails allows the developer to declare three different types of relationships using the following association declarations in pairs: `has_many`, `has_one`, `belongs_to` and `has_and_belongs_to_many`. To declare a one-to-many relationship, the `has_many` and `belongs_to` declarations are used (*e.g.*, lines 11 and 16 declare a one-to-many relationship between Profile and Photo). To declare a one to zero-or-one relationship, the `has_one` and `belongs_to` declarations are used (*e.g.*, lines 3 and 10 declare that a User is associated with zero or one Profile objects). Finally, to declare a many-to-many relationship two `has_and_belongs_to_many` declarations are used (*e.g.*, lines 2 and 7 to declare a many-to-many relation between User and Role).

B. Extending the Basic Relationships

Rails offers constructs to extend the basic relationship to express more complex relationships between objects. The first construct is the `:through` option, which can be set on either the `has_one` or `has_many` declaration. This option allows the developer to express transitive relationships. For example, lines 3, 10 and 11, 16 declare relationships between User and Profile, and between Profile and Photo. The `:through` option set on the association declaration on line 4 declares a relationship between User and Photo that is the composition of the ones between User and Profile, and Profile and Photo.

The second option is the `:conditions` option which allows the developer to create a relation between one class and the subset of another class. For instance, on line 13 the `:conditions` option is set on the relation between Profile and Video, denoting that Profile objects are only associated with Video files that satisfy the condition `"format='mp4'"`.

The third option is the `:polymorphic` option, which can is set on a `:belongs_to` declaration. This option creates a relation that multiple other classes can relate to, similar to the idea of interfaces in object-oriented programming. In the running example, line 25 sets up such a relation in the Tag

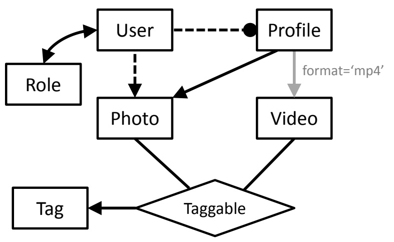


Fig. 2: The schema extracted from the data model in Figure 1.

class. The Photo and Video classes both connect to this relation by using the corresponding `:as` option in lines 17 and 22 (which can be set on either the `has_one` or `has_many`).

Finally, the `:dependent` option allows developers to model how to propagate the object deletion at the data model level. The `:dependent` option can be set to either `:delete` or `:destroy`, where `:delete` will propagate the delete to the associated objects, and no further, whereas `:destroy` will go into the class of the associated objects and propagate the delete further depending on the `:dependent` options set on its relations. On line 11 in Figure 1 we see that the `:dependent` option is set on the relationship between Profile and Photo. This means that when a Profile object is deleted, all associated Photo objects are also deleted. Since the `:dependent` option is set to `:destroy`, the delete will also be propagated to any relations in the Photo class with the `:dependent` option set.

III. Formalizing Data Models

In this section, we describe how Rails' data modeling constructs are formalized by our tool. A data model is formally defined as a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{C}, \mathcal{D} \rangle$ where $\mathcal{S}$ is the data model schema identifying the sets and relations of the data model, $\mathcal{C}$ is a set of relational constraints, and $\mathcal{D}$ is a set of dependency constraints [10], [11].

The schema is a tuple $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ that is made up of the object classes $\mathcal{O}$ and the relations $\mathcal{R}$ in the data model. The relations in the schema are specified as tuple consisting of the domain class, the relation name, the relation type and the range class: $\mathcal{R} \subseteq \mathcal{O} \times N \times T \times \mathcal{O}$.

For example, the schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ for the running example in Figure 1 consists of the object classes $\mathcal{O} = \{$User, Role, Profile, Photo, Video, Tag$\}$ and the object relations $\mathcal{R}$ contain seven tuples, one for each relation declared in Figure 1: User-Role, User-Profile, User-Photo, Profile-Photo, Profile-Video, Photo-Tag, Video-Tag. As an example, the tuple for the User-Photo relation is (User, User-Photo, (one, many, transitive, not-conditional, not-polymorphic), Photo).

Figure 2 shows a visual representation of the schema for the running example (which is automatically generated by our tool IDAVER). The nodes are the object classes and the edges are the object relations, which are depicted according to the type of relation.

In addition to the schema, a formal data model contains relational constraints, $\mathcal{C}$ that are imposed by their declarations. For example, lines 3 and 10 in Figure 1 declare a one to many relation between the Profile and Photo objects; this constraint would be represented as a relational constraint in $\mathcal{C}$. A formal data model also contains dependency constraints, $\mathcal{D}$, required for modeling delete dependencies. Given the state of the data model before a delete operation, these constraints specify how the deletion of an object and the usage of the `:dependent` options constrains the state of the data model after the delete operation.

Formalizing Verification Queries: In order to formalize verification queries, we define a data model instance as a tuple $I = \langle O, R \rangle$ where $O = \{o_1,\, o_2, \dots o_{n_O}\}$ is a set of object classes and $R = \{r_1,\, r_2, \dots r_{n_R}\}$ 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 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. Formally, $M \models A_S \Leftrightarrow \forall I = \langle O, R \rangle, I \models M \Rightarrow R \models A_S$,
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. Formally, $M \models A_B \Leftrightarrow \forall I = \langle O, R \rangle, \forall I' = \langle O', R' \rangle (I, I') \models M \Rightarrow (R, R') \models A_B$,
3) *state predicates* (denoted by P_S): These are properties we expect to hold in some instance of the data model. Formally, $M \models P_S \Leftrightarrow \exists I = \langle O, R \rangle, I \models M \wedge R \models P_S$,
4) *behavior predicates* (denoted by P_B): These are properties we expect to hold in some pair of instances that form a behavior of the data model. Formally, $M \models P_B \Leftrightarrow \exists I = \langle O, R \rangle, \exists I' = \langle O', R' \rangle), (I, I') \models M \wedge (R, R') \models P_B$.

IV. Property Templates

Manual specification of formal data model properties can be tedious and error-prone. Moreover, since our verification tool targets multiple theories for data model verification, manual specification of properties would require the user to learn the semantics and syntax of the input languages of all the solvers used by our tool, understand the specifications generated by our model extractor and translator, and then write the data model properties in the input language of the solver the user desires to use for verification. We believe that this would significantly reduce the usability of our tool. One of our contributions in this paper is to present a set of property templates that make the specification of data model properties easier. Since our tool integrates data model verification with different solvers in one framework, it can automatically translate the properties specified using these templates into the input language of the solver that the user chooses.

We have a total of eight property templates available for the user. These templates can easily be instantiated by the user for different classes and relations by providing the names of the object classes and relations as input.

We present the formal definitions of the eight property templates below. For the following, let M be the data model about which we are expressing the property. Let $I = \langle O, R \rangle$, $I' = \langle O', R' \rangle$ be data model instances, $o_A, o_B, o_C \in O$, $o'_A, o'_B \in O'$, $r_{A-B}, r_{B-C}, r_{A-C} \in R$, and $r'_{A-B}, r'_{B-C} \in R'$. Let $I \models M$ and $(I, I') \models M$.

I. *alwaysRelated* is used to express that objects from one class are always related to objects of another class. We formally define this template as

$$\textit{alwaysRelated}(o_A, o_B, r_{A-B}) \equiv \forall a \in o_A, \exists b \in o_B, (a, b) \in r_{A-B}$$

For example we can express the following property on the data model in Figure 1: *alwaysRelated*(Profile, User). This is saying that a Profile object should always be associated with a User object.

II. *multipleRelated* expresses the property that it is possible for the objects of one class to be related to more than one object of another class. Formally,

$$\textit{multipleRelated}(o_A, o_B, r_{A-B}) \equiv \exists a \in o_A, b_1, b_2 \in o_B, \\ b_1 \neq b_2 \wedge (a, b_1) \in r_{A-B} \wedge (a, b_2) \in r_{A-B}$$

In the running example, we can specify *multipleRelated*(Photo, Tag) to state that a Photo may be associated with more than one Tag.

III. *someUnrelated* is used to express that it is possible for an object of one class to not be related to any objects of another class. This template is defined formally as

$$\textit{someUnrelated}(o_A, o_B, r_{A-B}) \equiv \exists a \in o_A, \forall b \in o_B, (a, b) \notin r_{A-B}$$

For example, the property *someUnrelated*(User, Photo) means that it is possible to have a User without any Photos.

IV. *transitive* is the template used to express that one relation is the composition of two others. Formally,

$$\textit{transitive}(o_A, o_B, o_C, r_{A-B}, r_{B-C}, r_{A-C}) \equiv \\ \forall a \in o_A, c \in o_C, (a, c) \in r_{A-C} \iff \\ \exists b \in o_B, (a, b) \in r_{A-B} \wedge (b, c) \in r_{B-C}$$

For the running example, the property *transitive*(User, Profile, Photo) states that the relation between User and Photo is the composition of the relation between User and Profile, and User and Photo.

V. *noOrphans* applies to situations where objects can potentially be orphaned. This occurs when a class, o_B, has only one relation, i.e. it is connected to the schema graph via exactly one relation, r_{A-B}. In this case, when an element of class o_A is deleted it is possible that its associated elements in o_B may be *orphaned*—left without any connections to other objects. This property template is used to check for such scenarios. Formally,

$$\textit{noOrphans}(o_A, o_B, r_{A-B}) \equiv \forall a \in o_A, b' \in o'_B \\ a \notin o_A \implies \exists a' \in o'_A, (a', b') \in r'_{A-B}$$

As an example, we may desire to check *noOrphans*(Video, Tag) to make sure there are no orphaned Tags once a Video has been deleted.

VI. *noDangling* is the template used to express that when an object of one class is deleted, there are no objects of another class that are left with a dangling reference to this deleted object. Defined formally,

$$\textit{noDangling}(o_A, o_B, r_{A-B}) \equiv \forall a, a' \in o_A, b' \in o'_B \\ a \notin o_A \implies ((a', b') \in r'_{A-B} \implies a' \in o'_A)$$

For example, *noDangling*(Profile, Photo) expresses that when a Profile object is deleted, there are no Photo objects that have a dangling reference to a Profile object, i.e. no Photo object is related to a deleted Profile object.

VII. *deletePropagates* template is about making sure that when an object of one class is deleted, related objects in another class are also deleted. This template is formally defined as:

$$\textit{deletePropagates}(o_A, o_B, r_{A-B}) \equiv \forall a \in o_A, b \in o_B \\ a \notin o_A \implies ((a, b) \in r_{A-B} \implies b \notin o'_B)$$

For instance, we can say *deletePropagates*(Profile, Video), meaning that when a Profile object is deleted then the delete is propagated to all associated Video objects.

VIII. *noDeletePropagation* is the template used to express that when an object of one class is deleted, its associated objects from another class are NOT deleted. Formally,

$$\textit{noDeletePropagation}(o_A, o_B, r_{A-B}) \equiv \forall a \in o_A, b \in o_B \\ a \notin o_A \implies ((a, b) \in r_{A-B} \implies b \in o'_B)$$

For example, *noDeletePropagation*(User, Role) means that when a User is deleted, the associated Role should not be deleted.

V. iDAVER: An Integrated Data Model Verifier

iDAVER (Integrated DAta model VERifier) takes as input a set of model files from a Ruby on Rails 2.0 application and a set of properties in the form of templates (discussed in Section IV). The user can choose one of three verification options: 1) bounded verification with Alloy Analyzer 4, 2) unbounded verification with the SMT Solver Z3 4.3, and 3) unbounded verification with the first order logic theorem prover Spass 3.5.

iDAVER extracts a formal data model (discussed in Section III) from the Rails data model files, which is then translated into a specification in the language of the chosen solver. The properties specified (using templates) are also translated into the modeling language of the chosen solver and appended to the specification. Then the specification is fed into the solver, and the output of the solver is interpreted by iDAVER and shown back to the user. In cases where the output contains a satisfying or violating instance, iDAVER translates the output of the solver to an instance of the data model (in terms of sets and relations of the data model) before presenting it to the user.

A. Bounded Verification with Alloy

For bounded verification, iDAVER translates the formal data model it extracts into the Alloy language. To demonstrate how this translation works, consider the following Rails data model excerpt:

```
class User < ActiveRecord::Base
   has_one :profile
end
class Profile < ActiveRecord::Base
   belongs_to :user
end
```

This excerpt specifies a one to zero-or-one relation between User and Profile objects. Its translation to Alloy is given below:

```
sig Profile {}
sig User {}
one sig State {
   profiles: set Profile,
   users: set User,
   relation: Profile lone -> one User
}
```

The keyword `sig` is used in Alloy to define a set of objects. Thus, a `sig` is created for each class in the input Rails data model. In this example, a `sig` is declared for the Profile and User classes. We also create a State `sig`, which we use to define the state of a data model instance. Since we only need to instantiate exactly one State object when checking properties, we prepend the `sig` declaration with a multiplicity of `one`. The State `sig` contains fields to hold the set of all objects and related object pairs. In this example, the State `sig` contains three fields. The first is named `profiles` and is a binary relation between State and Profile objects. The field uses the multiplicity operator `set`, meaning 'zero or more'. In other words, the state of a data model instance may contain zero or more Profile objects. The State `sig` contains a similar field for User objects. Finally, the one to zero-or-one relation between Profile and User objects is translated as another field in the State `sig`. Named `relation`, it is defined to be a mapping between Profile and User objects. It uses the multiplicity operators `lone` and `one` to constrain the mapping to be between 'zero or one' Profile and 'exactly one' User object, respectively.

The translation of all Rails' data modeling constructs into Alloy is discussed in [10]. After iDAVER automatically translates the input data model and property template into an Alloy specification, it sends the specification to the Alloy Analyzer. Alloy Analyzer [5] is a bounded verification tool that converts verification queries to Boolean SAT problems and uses a SAT-solver to determine the result. iDAVER interprets this result and reports back to the user whether the data model property failed or verified. iDAVER also outputs satisfying instances for predicate properties that verify and counterexample instances for failing assertion properties.

B. Unbounded Verification with an SMT Solver

IDaVer can also be used to perform unbounded verification. It gives two options to do so: a Satisfiability Modulo Theories (SMT) solver or a First Order Logic (FOL) theorem prover. If the SMT solver is chosen to perform verification, IDaVer translates the formal data model into an SMT-LIB specification. The generated SMT-LIB specification is a formula in the theory of uninterpreted functions. For example, the translation of the data model excerpt (given earlier) is:

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

Types in SMT-LIB are declared using the `declare-sort` command. We use this command to declare types for User and Profile, as shown above. The relation is translated as an uninterpreted function. Uninterpreted functions are created in SMT-LIB using the `declare-fun` command. We use this command to declare an uninterpreted function name `relation` whose domain is Profile and range is User. Since functions can map multiple elements in the domain to the same element in the range, and we instead have a one to zero-or-one relation, we constrain the function to be one-to-one to obtain the desired semantics. This constraint is expressed using the `assert` command, above. Details for the complete translation of the all data modeling constructs in Rails are provided in [11].

Once the formal data model and property are automatically translated into SMT-LIB, IDaVer uses the SMT solver Z3 to determine the satisfiability of the generated formula. Based on the output of the SMT solver, IDaVer reports whether the property holds or fails. For failing assertions it also reports a data model instance as a counterexample, or a satisfying instance for predicate properties that verify.

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 [1]. In such cases IDaVer reports a “timeout”.

C. Unbounded Verification with a FOL Theorem Prover

Finally, IDaVer also performs unbounded verification using a FOL theorem prover. When this option is chosen, IDaVer translates the formal data model into first order logic axiom formulas. Next, it translates the property that is to be verified into a conjecture. Then the FOL theorem prover Spass is used to determine whether the property holds by checking if the axiom formulas imply the property.

To demonstrate how the translation is done to first order logic, we use the data model excerpt provided earlier. The result of its translation is given below:

```
1   list_of_symbols.
2     predicates[(relation, 2)].
3     sorts[Profile, User].
4   end_of_list.
5   list_of_formulae(axioms).
6     formula(forall([Profile(a)], not(User(a)))).
7     formula(forall([User(a)], not(Profile(a)))).
8     formula(forall([a, b], implies(
        relation(a, b),  and(Profile(a), User(b))))).
9     formula(forall([a, b1, b2], implies(
        and(relation(b1,a), relation(b2,a)), equal(b1,b2)))).
10    formula(forall([a, b1, b2], implies(
        and(relation(a,b1), relation(a,b2)), equal(b1,b2)))).
11    formula(forall( [Profile(a)],
        exists([b], relation(a, b)) )).
12 end_of_list.
```

Class types are translated into unary predicates (line 3); those denoting classes not related by inheritance are specified to be mutually exclusive (lines 6 and 7). Each relation is translated into a binary predicate (line 2) that can only be true if both elements satisfy their corresponding class type predicate (line 8).

The cardinality of relations is translated into additional axiom formulas. For example, the constraint that each User object is related to at most one Profile object is translated into the formula on line 9. For the constraint that each Profile object is associated with exactly one User object, we add two formulas: one similar to that on line 9 and one that says each Profile is related to at lease one User (lines 10 and 11, respectively).

There are additional data modeling constructs that are not covered in this example but are automatically translated by IDaVer. Polymorphic relations are translated like non-polymorphic relations, except that the elements are restricted to multiple types. A transitive relation between objects a and c, of classes A and C that goes through class B is defined as:

```
formula(forall( [a, c], implies(and(A(a), B(b)),
  equiv( through_relation(a, c),  exists( [b],
         and(B(b), subrelation1(a, b), subrelation2(b, c)))
  )  ))).
```

For conditional relations we introduce a unary predicate that represents that the condition holds, and add a formula that forces this predicate to hold on each object on the target side of the relation.

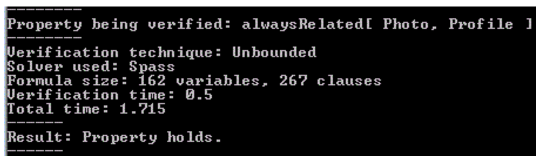
Delete dependencies introduce additional complexity to this basic translation. Two unary predicates are added: `Is_Deleted` and `Is_Destroyed`. They mark objects as deleted or destroyed, respectfully. A formula is added that states, for all objects for which `Is_Destroyed` is true, `Is_Deleted` is true by implication. In addition, all destroyed objects propagate the deleted or destroyed status to related objects, as defined by the dependencies in the Rails data model.

After the translation IDaVer calls the FOL theorem prover Spass. Since FOL is undecidable, it is possible that the theorem prover may not return a result. In such a case, IDaVer timeouts after the specified time limit. Otherwise, it interprets the results and reports whether the property is verified or not.

VI. A Case Study

We present a case study on an open source Rails application called LovdByLess. This is a social networking application with the usual features of users creating profiles, making friends, and leaving messages for friends. It also includes a forum where users can post and discuss topics. IDAVER takes as input the path of the directory containing the Rails data model files, and the name of the file containing the data model property(ies). The properties are expressed using the property templates discussed is Section IV. To make entry of properties as easy as possible, the templates only require the class names as input and the relations are inferred by IDAVER.

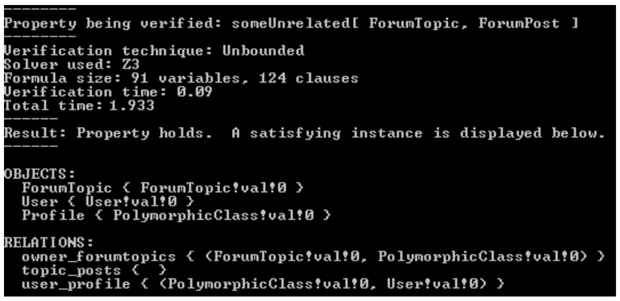
To start, let us say the user desires to verify the data model of the LovdByLess application by ensuring the property `alwaysRelated[Photo, Profile]` holds, meaning a Photo object is always associated with some Profile object. The user chooses to perform unbounded verification using Spass, the FOL theorem prover. Running IDAVER on these inputs gives the following result:



```
---------
Property being verified: alwaysRelated[ Photo, Profile ]
---------
Verification technique: Unbounded
Solver used: Spass
Formula size: 162 variables, 267 clauses
Verification time: 0.5
Total time: 1.715
------
Result: Property holds.
------
```

IDAVER outputs the name of the property being verified, the verification technique and solver used, the total time spent to obtain the result (including the taken to perform the translation into the input language of the solver), the time spent by the solver to perform the verification, and the size of the formula in terms of the number of variables and clauses in the specification. The formula size is solver-dependent: for SMT-LIB and Spass, variables are the number of types, functions, and quantified variables in the specification, and clauses are the number of asserts, quantifiers and operations; for Alloy this is the number of variables and clauses generated in the boolean formula generated for the SAT-solver used by Alloy. Finally, IDAVER outputs the result of the verification, which in this case is that the property holds on the data model.

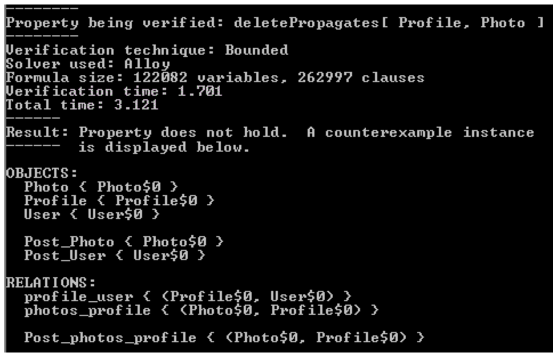
Next, let us change the solver to the SMT solver Z3, and verify a different property. `someUnrelated[ForumTopic, ForumPost]` checks that it is possible to have topics in a forum that does not have any posts. IDAVER outputs the following when run on these parameters[1]:



```
---------
Property being verified: someUnrelated[ ForumTopic, ForumPost ]
---------
Verification technique: Unbounded
Solver used: Z3
Formula size: 91 variables, 124 clauses
Verification time: 0.09
Total time: 1.933
------
Result: Property holds.  A satisfying instance is displayed below.
------

OBJECTS:
  ForumTopic < ForumTopic!val!0 >
  User < User!val!0 >
  Profile < PolymorphicClass!val!0 >

RELATIONS:
  owner_forumtopics < (ForumTopic!val!0, PolymorphicClass!val!0) >
  topic_posts <  >
  user_profile < (PolymorphicClass!val!0, User!val!0) >
```

This scenario demonstrates a benefit gained by using Z3 over Spass as the unbounded verification solver: although Spass tends to timeout less and be slightly faster than Z3 according to experiments we have done, Z3 formulates counterexample instances and satisfying instances (as in this example), which theorem provers are not designed to do.

In addition to unbounded verification, IDAVER can also perform bounded verification using Alloy Analyzer. Specifying bounded verification and using the default bound of twenty (meaning up to twenty objects of each class will be instantiated to check the property), we input the property `deletePropagates[Profile, Photo]` to check that when a Profile object is deleted, all associated Photo objects will also be deleted. IDAVER gives the following output[1]:



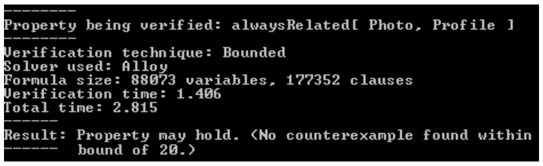
```
---------
Property being verified: deletePropagates[ Profile, Photo ]
---------
Verification technique: Bounded
Solver used: Alloy
Formula size: 122082 variables, 262997 clauses
Verification time: 1.701
Total time: 3.121
------
Result: Property does not hold.  A counterexample instance
------   is displayed below.

OBJECTS:
  Photo < Photo$0 >
  Profile < Profile$0 >
  User < User$0 >

  Post_Photo < Photo$0 >
  Post_User < User$0 >

RELATIONS:
  profile_user < (Profile$0, User$0) >
  photos_profile < (Photo$0, Profile$0) >

  Post_photos_profile < (Photo$0, Profile$0) >
```

Like Z3, Alloy also produces instances. IDAVER interprets the instances provided and translates them to a language-neutral and easy-to-understand form consisting of sets of objects and related object pairs. In the above example, IDAVER informs us that the property failed verification and provides us with a counterexample in the format just described. Since the user expected the property to hold, the counterexample is useful for understanding the cause of the error.

In this example, investigation into the application code reveals that all data related to a user, including Photos, should be deleted once a user's Profile is deleted. This can be enforced in the data model using the `:dependent` option in the Profile class on its relation with Photo, but currently this option is not set. Running the application demonstrates that this failing property does not manifest as an application error since the user interface of the application currently does not allow Profiles to be deleted, only to be deactivated. Thus this failing property is an example of a data model error (an error in the design of the data model) that does not show up as an application error because it is enforced in other parts of the application code.

Even though we performed bounded verification for this property, we were able to determine that the property failed. However this may not always be the case. In fact, the main disadvantage of bounded verification is that its results are not sound for verified assertions and failing predicates. For example, checking the `alwaysRelated[Photo, Profile]` property with Alloy Analyzer (which we checked earlier with Spass) gives the following result:

[1]The instances produced by the solvers were simplified to save space.



```
--------
Property being verified: alwaysRelated[ Photo, Profile ]
--------
Verification technique: Bounded
Solver used: Alloy
Formula size: 88073 variables, 177352 clauses
Verification time: 1.406
Total time: 2.815
------
Result: Property may hold. (No counterexample found within
------    bound of 20.)
```

Notice that IDAVER outputs that the property *may* hold since no counterexample was found within the bound. In this case unbounded verification gives a stronger verification result.

However, bounded verification has its own benefits. Recall from the previous section that unbounded solvers may timeout since satisfiability of FOL formulas and formulas in the theory of uninterpreted functions with quantification are known to be undecidable. This is the main benefit of having a tool that integrates both bounded and unbounded verification: in cases where unbounded solvers are unable to prove or disprove a property, IDAVER provides the user with the option to perform bounded verification to obtain an answer for the verification query with in a given bound.

Let us look at one final example. To check that deleting a Blog entry does not cause any Comment objects to have a dangling reference, we check the `noDangling[Blog, Comment]` property. Using Z3 to do unbounded verification, IDAVER returns that this property does not hold on the data model and provides a counterexample to help the user pinpoint the error. When the user runs the application and deletes a Blog entry, we see that in the database the associated Comments are left with a dangling reference to the deleted Blog entry. In the application, the user's main page contains a list of recent activity. This includes when someone has commented on that user's blog entry. The application sees there is a comment made for this user, but it cannot find the referenced blog entry. The application tries to make up for this error by returning an empty string. An application error occurs nonetheless since this empty string is displayed on the screen where text is expected. Using IDAVER, the user discovers a bug that can now be fixed in the data model by setting the `:dependent` option on the relation to Comment in the Blog class, so that all Comments are deleted when a Blog entry is deleted. This example demonstrates the importance of verifying data models and using the data modeling constructs available in Rails to enforce properties of data models. Using other parts of the application to enforce properties that should and can be upheld by the data model may lead to errors, such as this one.

VII. Related Work

Alloy Analyzer has been used for bounded verification of data models by others [2], [13]. However, translation to Alloy is not automated in these earlier works. Rubicon [8] is a tool based on Alloy that targets the verification of the Controller component in Ruby on Rails applications, whereas our work focuses on data model verification. Furthermore, these approaches focus only on bounded verification whereas our tool also supports unbounded verification. There has been some recent work on unbounded verification of Alloy specifications using SMT solvers [4], but to the best of our knowledge this approach has not been implemented.

There has been recent work on the specification and analysis of conceptual data models [12], [7]. 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.

The data model property inference and repair techniques presented in [9] are complementary to the contributions we present in this paper. In this paper, rather than trying to automatically synthesize the data model properties, we are focusing on providing templates that enable users to specify the data model properties at a high level.

The idea of using patterns to facilitate formal property specification was first proposed for temporal logic properties [3]. The property templates we present in this paper are not temporal and they are specific to data model analysis.

VIII. Conclusion

We presented a tool (IDAVER) for verifying data models of web applications written using the Rails framework. IDAVER integrates bounded and unbounded verification techniques, and supports property templates that simplify the task of property specification. IDAVER is applied directly on application code and does not require users to be familiar with any specialized modeling language or formal notation. We presented a case study demonstrating that IDAVER can be used on real-world web applications, and it can help developers in identifying errors in the web application data models.

References

[1] 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.

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

[3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. International Conference on Software Engineering (ICSE)*, pages 411–420, 1999.

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

[5] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Massachusetts, 2006.

[6] 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.

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

[8] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20)*, page 60, 2012.

[9] J. Nijjar and T. Bultan. Data model property inference and repair (under submission). http://www.cs.ucsb.edu/~bultan/publications/NB13submitted.pdf.

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

[11] J. Nijjar and T. Bultan. Unbounded data model verification using smt solvers. In *Proc. 27th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2012.

[12] 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.

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