Data Model Property Inference and Repair

Jaideep Nijjar, Tevfik Bultan
Department of Computer Science
University of California
Santa Barbara, CA
USA
{jaideepnijjar,bultan}@cs.ucsb.edu

ABSTRACT

Nowadays many software applications are deployed over compute clouds using the three-tier architecture, where the persistent data for the application is stored in a backend datastore and is accessed and modified by the server-side code based on the user interactions at the client-side. The data model forms the foundation of these three tiers, and identifies the set of objects stored by the application and the relations (associations) among them. In this paper, we present techniques for automatically inferring properties about the data model by analyzing the relations among the object classes. We then check the inferred properties with respect to the semantics of the data model using automated verification techniques. For the properties that fail, we present techniques that generate fixes to the data model that establish the inferred properties. We implemented this approach for web applications built using the Ruby on Rails framework and applied it to five open source applications. Our experimental results demonstrate that our approach is effective in automatically identifying and fixing errors in data models of real-world web applications.

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

Property inference, Data model repair, Automated verification, Web application modeling and analysis

1. INTRODUCTION

The software-as-a-service paradigm supported by cloud computing platforms has become a powerful way to develop software systems that are accessible everywhere and can store, access and manipulate large amounts of information. However, these benefits come with a cost: the increasing complexity of software applications. A typical software application nowadays is a complicated distributed system that consists of multiple components that run concurrently on multiple machines and interact with each other in complex ways via the Internet. As one would expect, developing such software systems is an error-prone task. Moreover, due to the distributed and concurrent nature of these applications, existing testing, static analysis and verification techniques are becoming ineffective. It is necessary to develop novel analysis techniques that focus on and exploit the unique characteristics of modern software applications.

Most modern software applications are developed using the three-tier architecture that consists of a client, a server and a backend datastore. The client-side code is responsible for coordinating the interaction with the user. The server-side code implements the business logic and determines the control flow of the application. The backend datastore stores the persistent data for the application. The interaction between the server and the backend datastore is typically managed using an object-relational mapping (ORM) that maps the object-oriented code at the server side to the relational database at the backend. The ORM makes use of a data model to accomplish this. A data model specifies the types of objects (*e.g.*, user, account, etc.) stored by the application and the relations among the objects (*e.g.*, the relation between users and accounts). A data model also specifies constraints on the data model relations (*e.g.*, the relation between two object types must be one-to-one). Since data models form the foundation of such applications, their correctness is of paramount importance.

In this paper, we focus on the correctness of data models in web applications. There are several popular frameworks for web application development such as Ruby on Rails, Zend for PHP, CakePHP, Django for Python, and Spring for J2EE. Web applications are globally accessible software systems that are in use all the time without any downtime for analysis or repair. So it is important that errors in web applications are both discovered fast and repaired fast. One way to achieve this is to increase the level of automation as much as possible in the analysis of such systems.

For most verification techniques and tools, the set of properties to be verified must be provided as an input to the verification process. The effectiveness of the verification process is highly dependent on the quality of the input properties. A verification tool cannot find an error in the input system

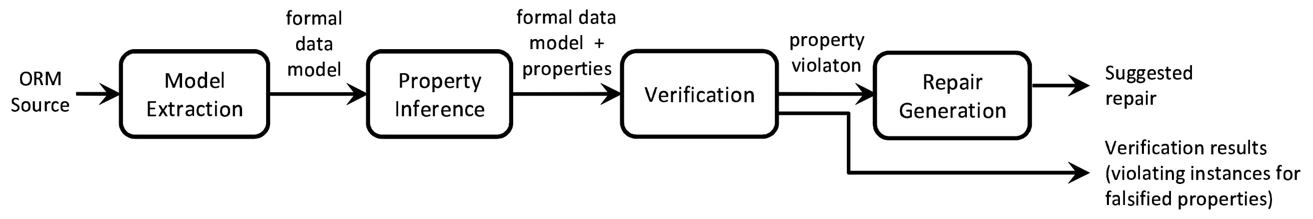

Figure 1: Data model analysis toolset.

if a property that exposes the error is not provided as input. Since manually writing properties is time-consuming, error-prone and lacks thoroughness, many errors can be missed during verification. Another disadvantage of the manual specification of properties is that it requires familiarity with the modeling language in which the properties are to be written, which is typically not the case for most developers.

In this paper, we propose novel techniques that automatically infer properties about the data model of web applications built using the three-tier architecture. The first step of our approach to data model property inference is extracting a formal data model from the ORM specification of the application. The extracted data model consists of a data model schema and a set of data model constraints. The data model schema is a directed and annotated graph representing the relations in the data model. We developed heuristics that explore the structure of this graph and look for a set of patterns. For example, if there are two alternative paths in the data model graph between two object classes, then in some cases we can infer that the relation that corresponds to the composition of the relations on one path should be equal to the relation that corresponds to the composition of the relations on the other path. As another example, if the deletion of an object might cause some other objects to become disconnected in the relation graph, then we might infer that the deletion of those objects should be dependent (i.e., the deletion of one object should automatically trigger deletion of the related objects). When we find matches to these patterns in the data model schema, we infer the corresponding properties.

Once the automatically inferred properties are generated, a data model verification technique (such as the ones presented in our prior work [19, 20]) can be used to determine if the properties inferred by analyzing the structure of the data model schema are actually enforced by the data model constraints. The verification techniques we use generate a counter-example data model instance for failing properties that demonstrates the violation; this aids in identifying the potential error in the data model. Finally, our approach includes techniques that automatically generate repairs for the properties that fail. These repairs are suggested modifications to the data model that establish the inferred properties.

Our techniques are applicable to ORMs in general, and we have implemented the approach for the Ruby on Rails (Rails for short) framework. The techniques have been implemented as a toolset for data model analysis, verification and repair. The architecture of the toolset is shown in Figure 1. The front end automatically extracts a formal data model from the ORM specification of the web application. The model extraction, property inference, verification and repair components are all integrated together and use the results from the prior stages to generate the results needed for the following stages of the analysis.

We used our toolset to analyze five open source Rails applications. Our results indicate that the integrated automated property inference, verification and repair approach is effective in discovering and eliminating errors in data models of real-world web applications.

Our contributions in this paper include: 1) Novel automated techniques for property inference that a) extract a data model schema from the ORM, b) investigate the structure of the generated data model schema and c) generate properties that are expected to hold in the data model. 2) The integration of automated property inference techniques with automated verification techniques in order to identify which of the inferred properties are valid based on the semantics of the data model. 3) Automated repair generation techniques that propose modifications to the data model so that the modified data model satisfies the properties that fail. 4) Implementation of the proposed techniques for analyzing Rails data models. 5) Experimental evaluation of the proposed techniques on five open source Rails applications.

The rest of the paper is organized as follows: Section 2 discusses data models. Section 3 presents the proposed data model property inference algorithms. Section 4 discusses automated verification of the inferred properties. Section 5 presents the automated repair generation techniques. Section 6 presents our experimental results. Section 7 discusses the related work and Section 8 concludes the paper.

2. DATA MODELS

In modern software applications that use the three-tier architecture, the data model serves as an abstraction layer between the application code and the backend datastore. The data model identifies the sets of objects stored by the application and the relations (also known as *associations*) among the objects. The object-relational mapping (ORM) handles the translation of the data model between the relational database view of the backend datastore and the object-oriented view of the application code. In this section we will first give an overview of the data model constructs supported by the Ruby on Rails framework (Rails for short) and later give a formalization of data model semantics.

The Rails framework uses an ORM called Active Records. Figure 2 shows a simple Active Records specification based on an open source Rails application called Tracks. This application allows users to create todo lists. Todos are organized by contexts (such as school, work, home), and todos can be tagged. A set of preferences are saved for each user. We will be using this data model example to explain the various declarations Active Records supports and how they

```
1   class User < ActiveRecord::Base
2        has_one :preference,
3                :conditions => "is_active=true"
4        has_many :contexts
5        has_many :todos
6   end
7   class Preference < ActiveRecord::Base
8        belongs_to :user
9   end
10  class Context < ActiveRecord::Base
11       belongs_to :user
12       has_many :todos, :dependent => :delete
13  end
14  class Todo < ActiveRecord::Base
15       belongs_to :context
16       belongs_to :user
17       has_and_belongs_to_many :tags
18  end
19  class Tag < ActiveRecord::Base
20       has_and_belongs_to_many :todos
21  end
```

Figure 2: A simplified data model based on a web application called TRACKS that manages todo lists.

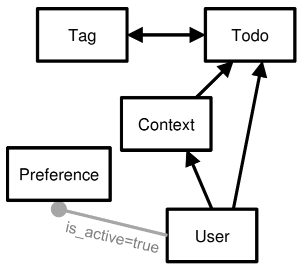

Figure 3: The data model schema extracted from the data model shown in Figure 2.

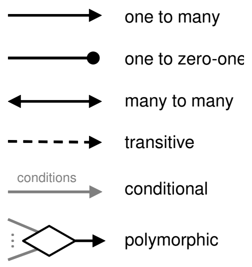

Figure 4: Graphical representations of relation types.

are used to express relations between objects. We then discuss how these relation declarations can be formalized as constraints in a formal data model.

2.1 Basic Relation Declarations

Rails supports three basic types of relations among objects: 1) *one to zero-one* relations, expressed using the `has_one` and `belongs_to` declarations, 2) *one to many* relations, expressed using the `has_many` and `belongs_to` declarations, and 3) *many to many* relations, expressed using the declaration `has_and_belongs_to_many`. For example, the `has_one` and `belongs_to` declarations of lines 2 and 8 in Figure 2 define a one to zero-one relation between the User and Preference classes. It declares that each User object must be associated with zero or one Preference object, and each Preference object must be associated with exactly one User object.

2.2 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 5 in Figure 2 that declares a relation between User and Todo objects. If we change line 5 as follows:

```
has_many :todos, :through => :contexts
```

then this would mean that the relation between User and Todo objects is the composition of the relations between User and Context objects (declared in lines 4 and 11) and Context and Todo objects (declared in lines 12 and 15).

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 on lines 2 and 3 in Figure 2, User objects are only related to a Preference object if its `is_active` field is `true`.

Rails also supports the declaration of polymorphic associations. A polymorphic relation is used when the programmer desires to use a single declaration to relate a class to multiple other classes. 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.

2.3 Delete Dependency

The final Rails construct used to express relations 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 themselves have relations with the `:dependent` option set and propagates the delete accordingly. In Figure 2 we see that the Context class has the `:dependent` option set for the relation with the Todo class (line 12). This means that when a Context object is deleted, the Todo objects that are associated with that Context will also be deleted.

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 analyze and infer properties about data models in order to discover errors. Below we describe the formal models we use in our analysis.

2.4 Formalizing Data Models

We formalize a data model 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.

The schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ identifies the object classes $\mathcal{O}$ and the relations $\mathcal{R}$ in the data model. In the schema each relation is specified as a tuple containing its domain class, its name, its type and its range class where $\mathcal{R} \subseteq \mathcal{O} \times N \times T \times \mathcal{O}$, N is a string denoting the name of the relation, and

$$T = \{\text{zero-one, one, many}\} \times \{\text{zero-one, one, many}\} \times \{\text{conditional, not-conditional}\} \times \{\text{transitive, not-transitive}\} \times \{\text{polymorphic, not-polymorphic}\}$$

is the set of relation types, which are a combination of type qualifiers denoting the cardinality of the domain and range of the relation, whether the relation is conditional or not, whether it is transitive or not and whether it is polymorphic or not. For example, the User-Preference relation defined in Figure 2 has the following type (one, one, conditional, not-transitive, not-polymorphic), indicating that it is a one to one relation that is conditional but not transitive and not polymorphic.

Not all combinations of these attributes are allowed in relation declarations. The types of relations must obey the following rules: 1) Only the following combinations of cardinalities are possible: many to many, one to many, many to one, zero-one to one, and one to zero-one. 2) A relation cannot be both polymorphic and transitive. 3) A many-to-many relation cannot be transitive nor polymorphic.

The schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ for the example shown in Figure 2 will consist of the following set of object classes $\mathcal{O} =$ {User, Preference, Context, Todo, Tag} and $\mathcal{R}$ will contain five tuples, one for each relation defined in Figure 2: User-Preference, User-Context, User-Todo, Context-Todo, Todo-Tag. For example, the tuple for the User-Preference relation will be: (User, User-Preference, (one, one-zero, conditional, not-transitive, not-polymorphic), Preference).

In Figure 3 we give a graphical representation of the schema extracted from the data model in Figure 2. The rectangular nodes in the graph correspond to the object classes and the edges represent the relations. The graphical representation of the edges differ based on the type of the relation that they represent, as explained in Figure 4.

The relational constraints, $\mathcal{C}$, in a formal data model express the constraints on relations that are imposed by their declarations. For example, lines 4 and 11 in Figure 2 declare a one to many relation between the User and Context objects. In order to formalize this cardinality constraint let us use o_U and o_C to denote the set of objects for the User and Context classes and r_{U-C} to denote the relation between User objects and Context objects. Then the constraint that corresponds to this relation is formalized as:

$$(\forall c \in o_C, \exists u \in o_U,\ (u, c) \in r_{U-C}) \wedge (\forall u, u' \in o_U, \forall c \in o_C, ((u, c) \in r_{U-C} \wedge (u', c) \in r_{U-C}) \Rightarrow u = u')$$

Semantics of all of the data model declaration constructs we discussed above other than the dependency constraints can be formalized similarly [20].

Formal modeling of the dependency constraints (denoted as D in the formal model) requires us to model the delete operation, which means that we have to refer to the state of the object classes and relations both before and after the delete operation. Consider the relation between Context and Todo objects. In order to model the delete operation we have to specify the set of Context objects, the set of Todo objects and the relation between the Context and Todo objects both before and after the delete operation (o_C, o'_C, o_T, o'_T, r_{C-T}, r'_{C-T}, respectively). Then we need to specify that when a Context object is deleted, the Todo objects related to that Context are also deleted. Formally:

$$o'_T \subseteq o_T \wedge o'_C \subseteq o_C \wedge r'_{C-T} \subseteq r_{C-T} \wedge (\exists c \in o_C, c \notin o'_C, (\forall c' \in o_C, c' \neq c \Rightarrow c' \in o'_C) \wedge (\forall t \in o_T, (c, t) \in r_{C-T} \Rightarrow t \notin o'_T) \wedge (\forall t \in o_T, (c, t) \notin r_{C-T} \Rightarrow t \in o'_T) \wedge (\forall c' \in o_C, \forall t \in o_T, ((c', t) \in r_{C-T} \wedge (c, t) \notin r_{C-T}) \Rightarrow (c', t) \in r'_{C-T}))$$

Note that modeling of cyclic delete dependencies would require the use of transitive closure. Our current framework does not handle cyclic delete dependencies.

3. PROPERTY INFERENCE

Below we present three heuristics for inferring three types of properties using the data model schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ defined in the previous section. These are the types of properties we encountered the most during our manual analysis of data models in our prior work [19, 20]. The heuristic algorithms take as input the data model schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ and output a list of properties. Each heuristic focuses on a sub-schema that contains only the relations that are relevant for the corresponding property type.

3.1 Delete Propagation

The first type of property we present is delete propagation. Our property inference algorithm for this type of property identifies when the deletion of an object should be propagated to objects related to that object. We denote this property as *deletePropagates*(r), where $r = (o, t, n, o') \in \mathcal{R}$ is a relation in the data model schema. The property *deletePropagates*(r) asserts that when an object from object class o is deleted then all objects from the object class o' that are related to the deleted object via the relation r are also deleted.

The heuristic for this property type first obtains a sub-schema by removing all relations in $\mathcal{R}$ that are transitive or many to many. This sub-schema is viewed as a directed graph, where an edge from o to o' corresponds to a one to many or one to zero-one relation, r, between classes o and o'. Such a sub-schema is given in Figure 5(a). Cycles in this graph are removed by collapsing strongly connected components to a single node. For the schema in Figure 5(a), nodes o_3 and o_4 are collapsed to a single node called c_1 in Figure 5(b). Next, each node in the schema is assigned a level that indicates the depth of a node in the graph. The root nodes(s) are those with no incoming edges and are at level zero. All other nodes are assigned a level that is one more than the maximum level of their predecessor nodes. The levels for the schema in Figure 5(a) are given in Figure 5(b). As can be seen, node o_1 is assigned level 0 since it has no incoming edges. The remaining nodes are assigned levels as just described.

The *deletePropagates* property is inferred if the difference in levels between the nodes a relation connects is not greater than one. The intuition here is that if the difference between the levels of the nodes is greater than one, then there could

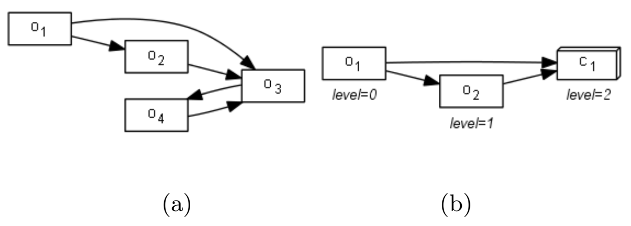

Figure 5: A sub-schema (a) and the corresponding acyclic graph (b) constructed during the Inference Algorithm for Delete Propagation.

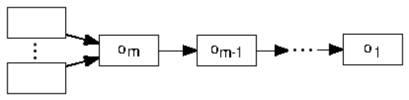

Figure 6: The pattern used for recognizing orphan chains.

be other classes between these two classes that are related to both of them and therefore propagating the delete could lead to inconsistencies between the relations. The complete algorithm for this heuristic is given in Algorithm 1.

Algorithm 1 Inference Algorithm for Delete Propagation

```
Input: Data model schema, $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$
Output: List of inferred properties
  Let $\mathcal{S}' = \langle \mathcal{O}, \mathcal{R}' \rangle$ be a data model schema where $\mathcal{R}' \subseteq \mathcal{R}$ only
  contains relations that are not transitive and not many to many.
  Let $S''$ be the directed acyclic graph obtained from $\mathcal{S}'$ by collapsing
  each strongly connected component in $\mathcal{S}'$ to a single node.
  for all nodes $x$ in $S''$ traversed in topological order do
      if node $x$ in $\mathcal{S}''$ has no predecessors then
          $level(x) = 0$
      else
          Let $x_1$,...,$x_n$ be the predecessors of $x$.
          $level(x) = max(level(x_1), \ldots, level(x_n)) + 1$
      end if
  end for
  For a node $c$ that corresponds to a strongly connected component,
  assign the $level$ of every class in the strongly connected component
  of $S'$ to be the $level$ of node $c$ in $\mathcal{S}''$.
  for all relations $r = (o, t, n, o')$ in $\mathcal{R}'$ do
      if $level(o') - level(o) = 1$ then
          Output $deletePropagates(r)$
      end if
  end for
```

3.2 Orphan Prevention

The next heuristic infers properties about preventing orphaned objects. An orphan object results after a delete operation if there is an object class related to a single other object class. An object becomes orphaned when the object it is related to is deleted but the object itself is not. Orphan chains can also occur, which begin with an object class that is related to a single object class, and continue with object classes that are related to exactly two object classes, one of which is the previous object class in the chain. Consider an object of the final class of a chain, such as o_{m-1} in Figure 6. When the object it is related to (of the class o_m) is deleted but the object itself is not, the entire chain of objects $(o_{m-1}, ..., o_1)$ becomes orphaned. We state the property which asserts that there are no orphans as $noOrphans(r)$, where $r = (o, t, n, o') \in \mathcal{R}$ is a relation. Specifically, this property asserts that deleting an object from object class o' does not leave any orphaned objects in class o, or orphan chains that begin with an object in class o.

The heuristic that infers this property looks for potential orphans or orphan chains by analyzing the directed graph that corresponds to the sub-schema which is obtained from the original data model schema by removing all relations in $\mathcal{R}$ that are not one to many or one to zero-one. The orphan prevention property inference algorithm is shown in Algorithm 2.

Algorithm 2 Inference Algorithm for Orphan Prevention

```
Input: Data model schema, $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$
Output: List of inferred properties
  Let $\mathcal{S}' = \langle \mathcal{O}, \mathcal{R}' \rangle$, where $\mathcal{R}' \subseteq \mathcal{R}$ contains only the relations that
  are either one to many or one to zero-one.
  for all classes $o \in \mathcal{O}$ with exactly one relation which is incoming,
  $r_1$, do
      Let $o'$ be the class $o$ is related to
      while $o'$ has exactly two relations, $r_1$ (outgoing), and another
  incoming, $r_2$, do
          Let $o := o'$
          Let $o' :=$ the class $o$ is related to by $r_2$
          Let $r_1 := r_2$
      end while
      Output $noOrphans(r_1)$
  end for
```

3.3 Transitive Relations

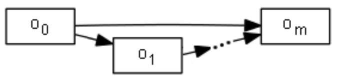

Figure 7: The pattern used for inferring transitive relations.

The final property inference heuristic is for detecting transitive relations. We state the transitive property as: $transitive(r_0, \ldots, r_m)$ where $m \geq 2$, $r_i = (o_i, t_i, n_i, o'_i) \in \mathcal{R}$ and $o'_i = o_{i+1}$ for $0 \leq i < m$, and $r_m = (o_0, t_m, n_m, o_m) \in \mathcal{R}$. This property asserts that the relation r_m is the composition of the relations $r_0, ..., r_{m-1}$.

The heuristic for this property defines a sub-schema by removing all relations in $\mathcal{R}$ that are polymorphic, transitive, conditional or many to many. The algorithm looks for paths of relations of length more than one. If there exists an edge connecting the first node in the path to the last node, then the algorithm infers that this edge should be a transitive relation. The intuition here is that if there are multiple ways to navigate relations between two classes, the composition of the relations corresponding to alternative ways of navigation should be equivalent. The pattern used for this heuristic is shown in Figure 7. Given that the path $o_0, o_1, ..., o_m$ is found, and there is also an edge between o_0 and o_m, the algorithm infers that this edge (o_0, o_m) should be transitive. The only exception is for paths that are of length exactly two. Then it is possible that the first edge in the path is the transitive relation so the algorithm outputs both possibilities. The complete algorithm for this heuristic is shown in Algorithm 3.

4. AUTOMATED VERIFICATION OF DATA MODELS

Algorithm 3 Inference Algorithm for Transitive Relations

```
Input: Data model schema, $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$
Output: List of inferred properties
  Let $\mathcal{S}' = \langle \mathcal{O}, \mathcal{R}' \rangle$, where $\mathcal{R}' \subseteq \mathcal{R}$ contains only relations that are either one to many or one to zero-one, and not polymorphic, transitive nor conditional.
  for all nodes $o_0 \in \mathcal{O}$ do
      for all pairs $(r_0, r_m)$ of outgoing edges from $o_0$ to distinct nodes $o_1$, $o_m$ do
          if there exists a path $p = (r_1, ..., r_{m-1})$ in $\mathcal{S}'$ from $o_1$ to $o_m$ then
              if $p$ is of length 2 then
                  Output $transitive(r_0, r_1, r_2) \vee transitive(r_2, r_1, r_0)$
              else
                  Output $transitive(r_0, ..., r_m)$
              end if
          end if
      end for
  end for
```

In order to check the properties generated by the property inference algorithms presented in Section 3, we integrate the automated verification techniques from our earlier work [19, 20] into our tool as shown in Figure 1. In this section, we overview these verification techniques.

We set up the discussion by formally defining data model instances and what it means for a data model instance to satisfy a given set of data model constraints. A data model instance is a tuple $\mathcal{I} = \langle O, R \rangle$ 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 $\mathcal{I} = \langle O, R \rangle$, we write $R \models \mathcal{C}$ to denote that the relations in R satisfy the constraints in C. Similarly, given two instances $\mathcal{I} = \langle O, R \rangle$ and $\mathcal{I}' = \langle O', R' \rangle$ we write $(R, R') \models \mathcal{D}$ to denote that the relations in R and R' satisfy the constraints in $\mathcal{D}$.

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

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

Given a data model $\mathcal{M} = \langle \mathcal{S}, \mathcal{C}, \mathcal{D} \rangle$, we define two 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; We denote that a data model satisfies an assertion as $\mathcal{M} \models A$ where:

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

For the three property types we discussed in the previous section, the *deletePropagates* and *noOrphans* properties are behavior assertions whereas the *transitive* property is a state assertion.

The data model verification problem is, given one of these types of properties, determining whether the data model satisfies the property. Two approaches for data model verification problem are summarized below.

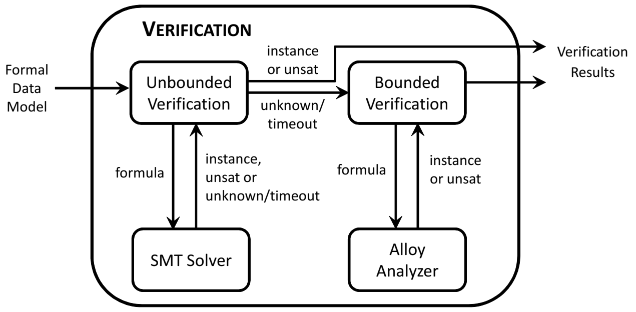

Figure 8: Verification component of the toolset.

4.1 Bounded Verification

One possible approach for data model verification is to use bounded verification where we check the property for instances within a certain bound [19]. The main idea is to bound the set of data model instances to a finite set, say $\mathcal{I}_k$, where $\mathcal{I} = \langle O, R \rangle \in \mathcal{I}_k$ if and only if for all $o \in O$ $|o| \leq k$. Then given a state assertion A_S, we can check the following condition:

$$\exists \mathcal{I} = \langle O, R \rangle, \mathcal{I} \in \mathcal{I}_k \wedge \mathcal{I} \models \mathcal{M} \wedge R \not\models A_S$$

Note that if this condition holds then we can conclude that the assertion A_S fails for the data model $\mathcal{M}$, i.e., $\mathcal{M} \not\models A_S$. However, if the condition does not hold, then we only know that the assertion A_S holds for the data model instances in $\mathcal{I}_k$.

An enumerative (i.e., explicit state) search technique is not likely to be efficient for bounded verification since even for a bounded domain the set of data model instances can be exponential in the number of sets in the data model. One bounded verification approach that has been quite successful is SAT-based bounded verification. The main idea is to translate the verification query to a Boolean SAT instance and then use a SAT solver to search the state space. Alloy Analyzer [15] is a SAT-based bounded verification tool for analyzing object-oriented data models. The Alloy language allows the specification of objects and relations as well as the specification of constraints on relations using first-order logic. In order to do SAT-based bounded verification of Rails data models, we use the automated translator described in [19] that translates Active Record ORM specifications to Alloy specifications. After this automated translation, we use the Alloy Analyzer for bounded verification of the data model properties we infer.

4.2 Unbounded Verification

To perform unbounded verification of data models, the technique we use is to convert the inferred property to a query about the satisfiability of formulas in the theory of uninterpreted functions. Given an Active Record ORM specification and a property (a state or a behavior assertion), we generate a formula in the theory of uninterpreted functions and then use a Satisfiability Modulo Theories (SMT) solver to determine the satisfiability of the generated formula [20]. Based on the output of the SMT solver, the verification tool reports whether the property holds or fails, and for failing assertions it also reports a data model instance as a counter-example.

Since this SMT-based verification approach does not bound the sizes of the object classes or the relations, if the verification tool reports that an assertion holds, it is guaranteed to hold for any data model instance. On the other hand, if the tool reports that a property fails, this means that the property fails to hold on the given data model and the counter-example data model instance generated by the tool is an instance for which the property does not hold.

In addition to returning unsatisfiable or satisfiable, an SMT solver may also return “unknown” or it may time-out since the quantified theory of uninterpreted functions is known to be undecidable [3]. So, when the call to the SMT-solver times out or returns “unknown” we switch to the SAT-based bounded verification approach. The entire verification approach is depicted in Figure 8 and forms the verification component of our tool architecture (shown in Figure 1).

5. PROPERTY REPAIR

After properties are inferred using the heuristics described in Section 3 and then verified as described in Section 4, our tool automatically generates data model repairs for the failed properties. These repairs show how the data model can be modified so that the failed property will hold in the repaired model. The repair rules we developed are discussed below.

5.1 Delete Propagation

First we explain the repair generation for the delete propagation property. If *deletePropagates*(r) fails for some relation $r = (o, t, n, o')$, this means that the data model is set up such that deleting an object of class o will not cause associated objects of o' to be deleted. In order to enforce this property in the data model, the `:dependent` option must be set on the `has_many` or `has_one` declaration corresponding to relation r in o's model. For example, when we run the Inference Algorithm for Delete Propagation (Algorithm 1) on the data model of the todo list application given in Figure 2, the *deletePropagates* property is generated for the relation between the User and Context classes. However, this property fails when we check it using the automated verification techniques discussed in the previous section. This means that when a user is deleted, the contexts created by the user are not deleted. In order to enforce this in the data model, the repair our tool generates sets the `:dependent` option on the relation with Context in the User model, i.e.

```
has_many :contexts, :dependent => :destroy
```

This will cause the deletion of User objects to be propagated to the associated Context objects. Note that the `:dependent` option is set to `:destroy` and not `:delete` since we want the delete to propagate to o''s associated objects. Otherwise there may be objects of another class with a dangling reference to the deleted associated object. In the running example, we observe that setting the `:dependent` option to `:delete` may result in Todo objects with a dangling reference to a deleted Context object. In order to prevent this inconsistency, the `:dependent` option is set to `:destroy` so that the Context model can propagate the delete to the desired relations. Figure 9 displays the data model of the todo list application with the automatically generated repair on line 4.

```
1   class User < ActiveRecord::Base
2        has_one :preference, :conditions => "is_active=true",
3               :dependent => :destroy
4        has_many :contexts, :dependent => :destroy
5        has_many :todos, :through => :contexts
6   end
7   class Preference < ActiveRecord::Base
8        belongs_to :user
9   end
10  class Context < ActiveRecord::Base
11       belongs_to :user
12       has_many :todos, :dependent => :delete
13  end
14  class Todo < ActiveRecord::Base
15       belongs_to :context
16       # line deleted
17       has_and_belongs_to_many :tags
18  end
19  class Tag < ActiveRecord::Base
20       has_and_belongs_to_many :todos
21  end
```

Figure 9: The data model from Figure 2 updated with the suggested repairs (in bold) generated by our tool.

5.2 Orphan Prevention

Next we present the repair for the orphan prevention property. When *noOrphans*(r) fails for a relation $r = (o, t, n, o')$, this mean that the data model is set up such that deleting an object of class o' will cause objects in class o to be orphaned, i.e. there will be objects of class o that will not be related to any other object. We can enforce this property in the data model by generating a repair that will delete the associated objects that would otherwise be orphaned. This is done by setting the `:dependent` option on the declaration corresponding to relation r in the model for o'. For orphan chains this is repeated down the chain, creating repairs for the declarations that associate a class with the next class in the chain.

For example, when we run the Inference Algorithm for Orphan Prevention (Algorithm 2) on the data model in Figure 2, a *noOrphans* property is generated which states that when a User is deleted no Preference objects should be orphaned. This property fails when we check it using automated verification, which means that when a user is deleted who has a preference, the preference object is orphaned. In order to enforce this property in the data model, a repair is generated that sets the `:dependent` on the relation with Preference in the User model, i.e.

```
has_one :preference, :conditions => "is_active=true",
                     :dependent => :destroy
```

This will cause the deletion of a User object to be propagated to the associated Preference. There are no more objects in this orphan chain so no further repairs will be generated. This suggested repair, as applied to the data model in Figure 2, is shown in Figure 9 on line 3.

5.3 Transitive Relations

Finally, we discuss the repairs for the failing transitive relations properties. When *transitive*$(r_0, \ldots, r_m)$ fails for some set of relations $r_0, \ldots, r_m$ it means that r_m is not the composition of the other m relations, as asserted in the property. To repair this property in the data model, we set the

`:through` option on the declaration corresponding to the relation $r_m = (o_m, t_m, n_m, o'_m)$ in o_m's data model. For instance, running the Inference Algorithm for Transitive Relations (Algorithm 3) on the example in Figure 2 infers the following *transitive* property: the relation between User and Todo should be the composition of the relations between User and Context, and Context and Todo. However, we again find out that this property fails using automated verification. In other words, the todos in the contexts created by a user may not be the same as the todos created by that user. In order to enforce this transitivity in the data model, a repair is generated which sets the `:through` option on the declaration in the User class that associates it with Todo:

```
has_many :todos, :through => :contexts
```

We also need to remove the `belongs_to :user` declaration in the Todo class since it becomes unnecessary when using the `:through` option. After this repair the relation between User and Todo will be the same as navigating the User-Context relation and then the Context-Todo relation. The data model for the todo list application after being modified by this repair is shown in Figure 9 (see lines 5 and 16).

There are two complications in the repair generation of the transitive relation property. For transitive properties with exactly three parameters, *transitive*(r_0, r_1, r_2), it is possible that r_0 is the transitive relation instead of r_2 so two repairs will be generated to let the user choose the one that is appropriate for fixing the failing property.

The other scenario is for transitive properties with more than three parameters. In Rails, one can only express that a relation is the composition of two others, not three or more others. Therefore, to repair a property such as *transitive*$(r_0, \ldots, r_m)$ with $m > 2$ and $r_i = (o_i, t_i, n_i, o'_i)$, the repair generator ensures that there are transitive relations between o_0 and o_i for $1 < i < m$. Otherwise it generates these transitive relations, and then sets the `:through` option on r_m so that it is the composition of r_{m-1} and the (possibly generated) relation between o_0 and o_{m-1}.

6. EXPERIMENTS

To evaluate the effectiveness of the techniques we proposed in this paper, we ran our tool on five open source Rails web applications. Our tool applies the property inference heuristics discussed in Section 3 to the Active Record files of the input web application. The properties inferred are sent to the next component of the tool which automatically translates the Active Record files to SMT-LIB and performs verification using the SMT-solver Z3 [24]. If any properties time out during verification (with a time out limit of five minutes), bounded verification is performed instead, using the Alloy Analyzer with a bound of 10, meaning at most 10 objects of each type are instantiated to check satisfaction of these properties.

The set of properties reported as failing by the tool are manually checked to determine which are data model errors as opposed to false positives. Data model errors are those properties that are not upheld by the data model despite its ability to do so. There are two categories of errors: properties that are not upheld in the application codebase thus causing an application error, and those that are not upheld in the data model but are enforced in other areas of the application (such as in the server-side code). Properties enforced in other areas of the application can cause application errors in the future since if the application code is changed in the future, it is possible that the property may no longer be upheld by the application. Of the remaining properties that failed which do not fall under these two categories, we have properties that failed because of the limitations of Rails constructs, and properties that are false positives, i.e. data model properties that were incorrectly inferred.

Table 1: Sizes of the Applications

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

6.1 The Applications

The five applications used in the experiments are listed in Table 1, along with their sizes in terms of lines of code, number of total classes, and number of data model classes. Descriptions of the applications are given below:

- LovdByLess (lovdbyless.com) is a social networking application with the usual features.
- Substruct (code.google.com/p/substruct) is an e-commerce application.
- Tracks (getontracks.org) is an application that helps users manage to-do lists, which are organized by contexts and projects.
- FatFreeCRM (fatfreecrm.com) is a light-weight customer relations management software.
- OpenSourceRails(OSR) (opensourcerails.com) is a project gallery that allows users to submit, bookmark, and rate projects.

6.2 Inference and Verification Results

The results of running our tool on the five applications are given in Table 2. For each application and type of property, it displays the number of properties that were inferred by the tool, the number that failed during verification, and the number that timed out during unbounded verification. A total of 145 properties were inferred, of which 93 failed and 3 timed out during unbounded verification. The three properties that timed out were shown to fail using bounded verification, giving a total of 96 failing properties. We manually investigated each of the failing properties to determine which correspond to data model errors. These results are also summarized in Table 2.

For example, a *noOrphans* property that was inferred and failed verification (i.e. fails to hold on the data model) is in the OSR application. In this application, Projects can be rated by users and the property that was inferred states that when a Project is deleted, the associated ProjectRatings should not be orphaned. This property fails, meaning it is not upheld by the data model. Manual inspection shows that it should be. Thus, this property is a data model error. However this property does not manifest as an error in the overall application since the user interface does not allow projects to be deleted. Nevertheless this indicates a potential application error which can be exposed if the application is later changed to allow project deletion. The repair generated for this error suggests setting the `:dependent` option on the declaration in the Project class that relates Projects to ProjectRatings so that any associated ProjectRatings are deleted along with a Project instead of being orphaned. This will ensure that the property holds whether it or not other parts of the application upholds it.

There are also a category of properties that are data model and application errors. These properties are those that fail to hold not only in the data model but the entire application as well. For instance, in Tracks a *deletePropagates* property was inferred that stated deleting a Context should delete any associated RecurringTodos. This property is not upheld in the data model. Further, it is not enforced in the application, so when a context is deleted and then the recurring todo is edited that was associated with that context, the application crashes when it cannot find the associated context. This is an example of a data model and application error.

Properties that fail verification are not necessarily errors. For example, a *transitive* property that failed was for LovdByLess, which has forums in which users are allowed to create topics and post messages inside the forum topics. The property inferred states that the relation between User and ForumPost is the transitive between the relations between User and ForumTopic, and ForumTopic and ForumPost. Manual analysis shows that the this relation should not be transitive due to the semantics of the application. It is not necessary that users must post to forum topics that they created, as transitivity requires. Thus, this failing property is classified as a false positive.

Properties may also fail due to limited expressiveness in Rails constructs. For instance, in FatFreeCRM accounts can be created for each customer, and multiple contacts can be associated with each account. A *deletePropagates* property that was inferred for this application stated that the deletion of an Account should propagate to the associated Contacts. However, in this application it is valid for there to be contacts that are not associated with any accounts. Hence the relationship that was desired here was a zero-one to many, not a one to many. Therefore this property is a failure due to limitations in Rails' expressiveness.

As an example of a failure due to a different Rails limitation there is a *deletePropagates* property that failed in Substruct which stated that deleting a Country deletes any associated Addresses. However, the Country table holds a list of all countries in the world which should never be deleted, nor does the user interface allow this. Thus, the inability to declare the Country model as undeletable causes this property to fail.

Of the 145 properties inferred by our tool for the five web applications we analyzed, 49 properties (33.8%) hold on the given data model, 63 of them (43.4%) fail and correspond to data model errors, 9 of them (6.2%) fail due to Rails limitations, and 24 of them (16.6%) fail and correspond to false positives. The fact that we are able to identify 63 data model errors in five web applications indicates that data model errors are prevalent in web applications and web application developers are not using advanced features of ORMs effectively. In addition to identifying errors in data models, we are able to show developers how to fix their data model using automated repair generation.

6.3 Performance

Our experiments included taking performance measurements as an additional indicator of the effectiveness of our approach. Specifically, we measured the time it took for the inference and verification of each property, as well as the formula size produced by the verification tools. These values were averaged over the properties for each property type per application. The results are summarized in Table 3. For the unbounded tool the formula size measures the SMT-LIB specification produced for the property. Here, the number of variables are the number of sorts, functions and quantified variables in the specification, and the number of clauses are the number of asserts, quantifiers and operations. For the bounded tool, the formula size reports the number of clauses and variables created by Alloy's SAT translation. The time taken for repair generation is not reported in Table 3 since it is almost zero for all properties.

We observe that the *transitive* properties had the longest inference time and the *deletePropagates* properties had the shortest inference time. The longest inference time was 0.2 seconds on average, with the exception of transitive properties for the LovdByLess application which had an average inference time of 1.5 seconds. The longest unbounded verification time was only 0.65 seconds on average. Even for the properties that timed out and bounded verification was used instead, we see that figures are reasonable, where 2.359 seconds was the longest bounded verification time. In summary, our approach is not only able to find errors effectively, it does so efficiently.

7. RELATED WORK

There has been recent work on specification and analysis of conceptual data models [22, 17]. These efforts follow the model-driven development approach whereas our approach is a reverse engineering approach that automatically extracts a data model from an existing application and analyzes it.

There are also earlier results on the formal modeling of web applications focusing on state machine-based formalisms to capture the navigation behavior (such as [14, 1, 13]). In contrast to this line of work, we focus on analysis of the data model rather than the navigational aspects of web applications.

Automated discovery of likely program invariants by observing runtime behaviors of programs has been studied extensively [8, 9, 10]. There has also been extensions of this style of analysis to inference of abstract data types [12]. Instead of using observations about the runtime behavior, we analyze the static structure of the data model extracted from the ORM specification to infer properties. Static verification of inferred properties has been investigated earlier [21]. Unlike these earlier approaches we are focusing on data model verification in web applications.

There has been earlier work on automatically repairing data structure instances [6, 5, 7, 16]. In this paper we are not focusing on generating code for fixing data model properties during runtime. Instead, we generate repairs that modify the data model declarations that fix the data model for all possible executions. Moreover, we focus on data model verification in web applications based on ORM specifications which is another distinguishing feature of our work.

There has been prior work on the verification of data models; these works present bounded verification approaches us-

Table 2: Inference and Verification Results

Application	Property Type	Inferred	Timeout	Failed	Data Model and Application Errors	Data Model Errors	Failures Due to Rails Limitations	False Positives
LovdByLess	deletePropagates	13	0	10	1	9	0	0
	noOrphans	0	0	0	0	0	0	0
	transitive	1	0	1	0	0	0	1
Substruct	deletePropagates	27	0	16	1	3	5	7
	noOrphans	2	0	1	0	1	0	0
	transitive	4	0	4	0	1	0	3
Tracks	deletePropagates	15	0	6	1	1	3	1
	noOrphans	1	0	1	0	0	0	1
	transitive	12	0	12	0	7	0	5
FatFreeCRM	deletePropagates	32	1	19	0	18	1	0
	noOrphans	5	0	0	0	0	0	0
	transitive	6	2	6	0	0	0	6
OSR	deletePropagates	19	0	12	0	12	0	0
	noOrphans	1	0	1	0	1	0	0
	transitive	7	0	7	0	7	0	0
Total	deletePropagates	106	1	63	3	43	9	8
	noOrphans	9	0	3	0	2	0	1
	transitive	30	2	30	0	15	0	15

Table 3: Inference and Verification Performance

Application	Property Type	Inference Time (s)	Verification Time (s) Unbounded	Verification Time (s) Bounded	Formula Size (clauses) Unbounded	Formula Size (clauses) Bounded	Formula Size (variables) Unbounded	Formula Size (variables) Bounded
LovdByLess	deletePropagates	0.002	0.057	-	47.0	-	16.8	-
	noOrphans	0.012	-	-	-	-	-	-
	transitive	1.512	0.024	-	30.5	-	20.5	-
Substruct	deletePropagates	0.000	0.138	-	39.0	-	15.1	-
	noOrphans	0.002	0.083	-	31.3	-	13.7	-
	transitive	0.215	0.081	-	23.6	-	17.8	-
Tracks	deletePropagates	0.002	0.031	-	31.9	-	13.3	-
	noOrphans	0.013	0.372	-	28.0	-	12.0	-
	transitive	0.098	0.050	-	11.6	-	17.4	-
FatFreeCRM	deletePropagates	0.001	0.033	2.359	84.9	465822	21.8	197307
	noOrphans	0.026	0.651	-	124.8	-	29.8	-
	transitive	0.126	0.053	1.105	99.1	71490	31.9	36658
OSR	deletePropagates	0.001	0.060	-	30.3	-	12.6	-
	noOrphans	0.011	0.033	-	27.0	-	12.0	-
	transitive	0.064	0.061	-	12.7	-	17.1	-
Average		0.139	0.123	1.732	44.4	268656	17.986	116982.5

ing the Alloy Analyzer [4, 23]. However, the translation to Alloy is not automated in these earlier efforts. Alloy has also been used for discovering errors in web applications related to browser and business logic interactions [2] which is a different class of errors than the ones we focus on in this paper.

Rubicon [18] is tool for verification of Controller (application logic) in Rails applications, whereas our work focuses on data model analysis. Further, we propose techniques to automatically infer properties, whereas [18] requires manual specifications to be added to the code. Finally, Rubicon is limited to bounded verification (using Alloy Analyzer), whereas we perform both unbounded and bounded verification. There has been some other recent work on unbounded verification of Alloy specifications using SMT solvers [11], but to the best of our knowledge this approach has not been implemented yet.

In this paper we use results from our earlier work on verification of data models [19, 20]. The focus of this work is different than these earlier results. Our earlier papers present techniques for performing bounded and unbounded verification of data models, whereas this paper presents property inference and repair techniques. Property inference and repair problems were not addressed in our earlier papers and thus are novel contributions of this paper. Also, the property inference and repair techniques presented in this paper use the data model schema which is not defined nor used in our earlier papers.

8. CONCLUSIONS

In this paper we presented techniques for property inference and repair for data models that are used in web applications built using the three-tier architecture. We first extract a formal data model from the object-relational mapping of a given application. The formal data model consists of a schema and a set of constraints. We analyze the structure of the relations in the data model schema to infer properties. Next we use automated verification techniques to check if the inferred properties hold on the data model. For failing properties we generate repairs that modify the data model in order to establish the failing properties. Our experimental results demonstrate that the proposed approach is effective in finding and repairing errors in real-world web applications.

9. REFERENCES

[1] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *Proceedings of*

the 19th IEEE International Conference on Automated Software Engineering (ASE), pages 100–109, 2004.

[2] B. Bordbar and K. Anastasakis. MDA and analysis of web applications. In *Proceedings of the VLDB Workshop on Trends in Enterprise Application Architecture (TEAA)*, pages 44–55, 2005.

[3] R. E. Bryant, S. M. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, pages 470–482, 1999.

[4] A. Cunha and H. Pacheco. Mapping between Alloy specifications and database implementations. In *Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 285–294, 2009.

[5] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 233–244, 2006.

[6] B. Demsky and M. C. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 176–185, 2005.

[7] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 64–73, 2007.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[11] A. A. E. Ghazi and M. Taghdiri. Relational reasoning via SMT solving. In *Proceedings of the 17th International Symposium on Formal Methods (FM)*, pages 133–148, 2011.

[12] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 255–265, 2006.

[13] 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 *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 235–244, 2010.

[14] M. Han and C. Hofmeister. Relating navigation and request routing models in web applications. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 346–359, 2007.

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

[16] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 620–624, 2009.

[17] M. J. McGill, L. K. Dillon, and R. E. K. Stirewalt. Scalable analysis of conceptual data models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 56–66, 2011.

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

[19] J. Nijjar and T. Bultan. Bounded verification of Ruby on Rails data models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77, 2011.

[20] J. Nijjar and T. Bultan. Unbounded data model verification using SMT solvers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 210–219, 2012.

[21] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[22] Y. Smaragdakis, C. Csallner, and R. Subramanian. Scalable satisfiability checking and test data generation from modeling diagrams. *Automated Software Engineering*, 16(1):73–99, 2009.

[23] L. Wang, G. Dobbie, J. Sun, and L. Groves. Validating ORA-SS data models using Alloy. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 231–242, 2006.

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