Data Model Property Inference, Verification and Repair for Web Applications

JAIDEEP NIJJAR, University of California, Santa Barbara
IVAN BOCIĆ, University of California, Santa Barbara
TEVFİK BULTAN, University of California, Santa Barbara

Most software systems nowadays are web-based applications that 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 sets of objects (object classes) and the relations among them (associations among object classes) stored by the application. In this paper, we present a set of property patterns to specify properties of a data model, as well as several heuristics for automatically inferring them. We show that the specified or inferred data model properties can be automatically verified using bounded and unbounded verification techniques. For the properties that fail, we present techniques that generate fixes to the data model that establish the failing properties. We implemented this approach for web applications built using the Ruby on Rails framework and applied it to ten 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

General Terms: Languages, Verification

Additional Key Words and Phrases: Data models, web applications, automated verification, automated repair

1. INTRODUCTION

Software applications are migrating from desktops to computer clouds. 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, while saving the end users from the hassles of software installation, configuration management, version updates and security patches. 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, and due to the increasing use of scripting languages, existing testing, static analysis and verification techniques are becoming ineffective.

Yet another challenge is the fact that modern software applications are globally accessible systems (via web browsers or mobile application front-ends) that are in use all the time without any downtime for analysis or repair. So it is important that errors in

This work is supported by the National Science Foundation, under grant CCF 1117708.

DOI:http://dx.doi.org/10.1145/0000000.0000000

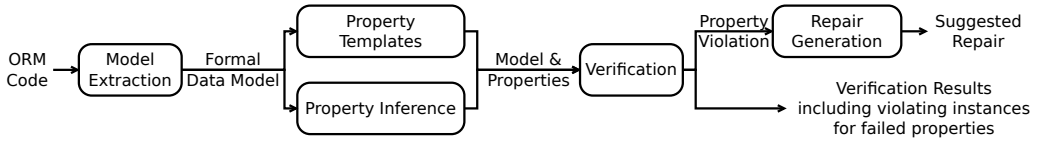

Fig. 1: Data model analysis toolset.

these applications are both discovered and repaired quickly. In order to address these challenges, it is necessary to develop novel techniques that focus on and exploit the unique characteristics of modern software applications while increasing the level of automation in analysis, verification and repair of such systems.

One common characteristic of modern software applications is the fact that they are developed using the three-tier architecture that consists of a client, a server and a backend datastore component. 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 keeps 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. To accomplish this, ORMs use the concept of a data model that specifies the types of objects (the object classes, e.g., user, account, etc.) stored by the application, the relations among the objects (the associations among the object classes, e.g., the association between users and accounts), and the constraints on the data model relations (e.g., the association between the users and accounts must be one-to-one).

Since data models form the foundation of modern software applications, their correctness is of paramount importance. For example, ORM specification correctness has been studied previously [McGill et al. 2011; Smaragdakis et al. 2007; 2009], as well as database schema correctness [Cunha and Pacheco 2009] which is related since ORM tools generate database schemas. In addition, bugs found by data model verification have been acknowledged and immediately repaired by developers [Bocic and Bultan 2014] indicating the importance of such bugs.

Most modern software development frameworks such as Ruby on Rails, Zend for PHP, CakePHP, Django for Python, and Spring for J2EE use the Model-View-Controller (MVC) pattern [Krasner and Pope 1988] which separates the data model (Model) from the user interface logic (View) and the control flow logic (Controller). The modularity and separation of concerns principles imposed by the MVC pattern makes automated extraction of the data model possible, and provides opportunities for developing customized verification and analysis techniques, which we exploit in this paper by presenting techniques for specification, verification and repair of data model properties.

For most verification techniques and tools, the set of properties to be verified must be provided as an input to the verification process, and the effectiveness of the verification process is highly dependent on the quality of the input properties. We present two approaches to facilitate property specification. First, we provide a set of property templates that cover a wide range of common properties of data stores (e.g., that an object of a certain class cannot exist in the absence of a related object from another class) and are easy to manually instantiate for a given data store [Nijjar et al. 2013]. Second, we present heuristics for automatic inference of properties about the data model of a given application [Nijjar and Bultan 2013].

Once we have a suite of properties, we use both bounded and unbounded data model verification techniques [Nijjar and Bultan 2011; 2012] to determine if the given properties are actually enforced by the data model constraints that are extracted from the ORM code. For failing properties, our method automatically generates a counter-example data model instance that demonstrates the violation which aids in identifying the potential error in the data model. Finally, we present techniques that automatically generate repairs for the properties that fail. These repairs are suggested modifications to the ORM code that establishes the inferred properties.

Our approach has its limitations. It is only applicable to properties that are expressible using templates that we provide. We do not analyze data model methods, but only focus on analysis of static association declaration constructs. We are not able to analyze unbounded data models with cyclic destroy dependencies, but we can use bounded analysis in such cases. Our experiments show that, even with these limitations, our approach is able to identify data model errors in real-world applications.

The high level structure of our approach 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 generation 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. In addition to automatically inferred properties, users can manually specify additional data model properties using property templates. The tool implementing our approach is called iDaVer and is available for download at http://www.cs.ucsb.edu/~vlab/idaver/.

The rest of the paper is organized as follows: Section 2 discusses the data models and their formalization. Section 3 discusses the property patterns and the automated property inference heuristics. Section 4 presents the techniques for automated verification of specified or 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 MODEL

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 (i.e., object classes) stored by the application and the relations (i.e., 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 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 employs ORM via a library called ActiveRecord. While our current toolset supports ActiveRecord only, other ORM libraries share many of the same features as ActiveRecord. Hence, the techniques we present in this paper are applicable to other ORM libraries.

Below we give an overview of Rails' data modeling features using a running example. Figure 2 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.

2.1. Basic Relation Declarations

Rails allows the developer to declare three different types of relations with different cardinality constraints using the `has_many`, `has_one`, `belongs_to` and

```
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, :conditions => "format='mp4'"
13  end
14  class Photo < ActiveRecord::Base
15    belongs_to :profile
16    has_many :tags, :as => :taggable
17  end
18  class Video < ActiveRecord::Base
19    belongs_to :profile
20    has_many :tags, :as => :taggable
21  end
22  class Tag < ActiveRecord::Base
23    belongs_to :taggable, :polymorphic => true
24  end
```

Fig. 2: A data model example

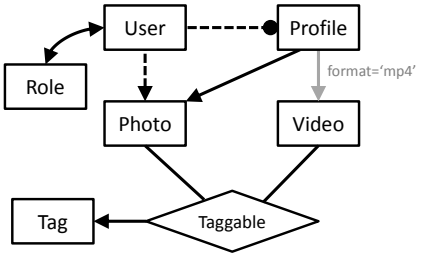

Fig. 3: The data model schema extracted from the data model shown in Figure 2.

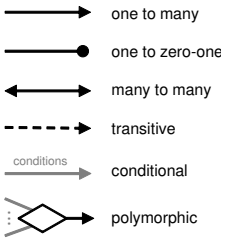

Fig. 4: Graphical representations of relation types.

has_and_belongs_to_many association declarations in pairs: 1) *one-to-zero-one* relations, expressed using the has_one and belongs_to declarations, (e.g., lines 3 and 10 in Figure 2 declare that a User is associated with zero or one Profile objects), 2) *one-to-many* relations, expressed using the has_many and belongs_to declarations (e.g., lines 11 and 15 declare a one-to-many relation between Profile and Photo), and 3) *many-to-many* relations, expressed using the declaration has_and_belongs_to_many (e.g., lines 2 and 7 to declare a many-to-many relation between User and Role).

In order to demonstrate that these types of relations can also be specified in other ORMs, we provide a simple syntax comparison between three different ORM libraries (ActiveRecord, Hibernate, and DjangoORM) in Figure 5, where objects of class Foo is associated with objects of class Bar with different cardinality constraints.

2.2. Extensions

Rails offers constructs to extend the basic relation declarations discussed above to express more complex relations between objects. The first construct we would like to discuss 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 relations. For example, lines 3, 10 and 11, 15 declare relations between User and Profile, and between Profile and Photo. The :through option set on the association declaration on line 4 declares a

Cardinality	ActiveRecord	Hibernate	DjangoORM

One to Zero or One	class Foo
	  has_one :bar
	end
	
	class Bar
	  belongs_to :foo
	end
		public class Foo {
		  @OneToOne(mappedBy = "bar")
		  public Bar getBar() { ... }
		}
		
		public class Bar {
		  @OneToOne(nullable = false)
		  public Foo getFoo() { ... }
		}
			class Foo(models.Model)
			  # associated Bar can be queried
			  # using the bar() method
			
			class Bar(models.Model)
			  foo = models.OneToOneField(Foo,
			  null=true,
			  primary_key=true)

One to Many	class Foo
	  has_many :bar
	end
	
	class Bar
	  belongs_to :foo
	end
		public class Foo {
		  @ManyToOne(mappedBy = "bar")
		  public Set<Bar> getBars() { ... }
		}
		
		public class Bar {
		  @ManyToOne
		  @JoinColumn(name = "foo_id",
		  nullable = false)
		  public Foo getFoo() { ... }
		}
			class Foo(models.Model)
			  # associated Bars can be queried
			  # using the bar_set() method
			
			class Bar(models.Model)
			  foo = models.ForeignKey(Foo)

Many to Many	class Foo
	  has_and_belongs_to_many :bar
	end
	
	class Bar
	  has_and_belongs_to_many :foo
	end
		public class Foo {
		  @ManyToMany(...)
		  @JoinTable(...)
		  public Set<Bar> getBars() { ... }
		}
		
		public class Bar {
		  @ManyToMany(mappedBy = "bars")
		  public Set<Foo> getFoos() { ... }
		}
			class Foo(models.Model)
			  # associated Bars can be queried
			  # using the bars() method
			
			class Bar(models.Model)
			  foo = models.ManyToManyField(Foo)

Fig. 5: ORM Library Syntax Comparison

relation between User and Photo that is the composition of the ones between User and Profile, and Profile and Photo.

The second construct 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 12 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'"`.

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 that 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. In the running example, line 23 sets up such a relation in the Tag class. The Photo and Video classes both connect to this relation by using the corresponding `:as` option in lines 16 and 20.

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 2 we see that the `:dependent` option is set on the relation 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.

All these constructs together form the essence of Rails data models specified using the ActiveRecord. As we discussed above, the basic constructs are also supported by other ORMs such as Hibernate and DjangoORM. ActiveRecord is the most expressive among these ORM libraries, and allows declaration of complex relations using the options we discussed above. However, many features of ActiveRecord are available in other ORMs as well. For example, Hibernate supports delete propagation in a man-

ner similar to `:dependent => :destroy` option of ActiveRecord. All three ORMs support polymorphism and multiple inheritance in different ways (polymorphic associations in Rails, `@MappedSuperclass` annotations in Hibernate, the `parent_link` option in DjangoORM). The `:condition` option of ActiveRecord is unavailable in Hibernate or DjangoORM, and DjangoORM does not have a feature similar to ActiveRecord's `:dependent` option. These missing features can be implemented as extensions to these frameworks.

The approach we present in this paper is applicable to other ORMs such as Hibernate and DjangoORM. Adapting our approach to these other frameworks would require implementation of new data model extraction modules to support these languages. However, the formal data model and all the analysis techniques would be applicable without modifications. Also, the repair generator would need to be tailored to the syntax and constructs of these languages.

2.3. Formalizing Data Models

Our verification framework first extracts a formal data model from the ORM code of a given web application based on the ActiveRecord class and association declarations that we discussed above. 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, and whether the relation is conditional, transitive or polymorphic. For example, the Profile-Video relation defined in Figure 2 has the type (one, many, conditional, not-transitive, not-polymorphic), indicating that it is a one to many relation that is conditional but not transitive or 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 or polymorphic.

The schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ for the running example in Figure 2 consists of the object classes $\mathcal{O} = \{\text{User, Role, Profile, Photo, Video, Tag}\}$ and the object relations $\mathcal{R}$ contain seven tuples, one for each relation declared in Figure 2: 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 3 shows a visual representation of the schema for the running example (which is automatically extracted from the ORM code by iDaVer). The nodes are the object classes and the edges are the object relations. The graphical representation of the edges differs based on the type of the relation that they represent, as explained in Figure 4.

The relational constraints, $\mathcal{C}$, express the constraints that are imposed by the relation declarations. For example, lines 12 and 19 in Figure 2 declare a one to many relation between the Profile and Video objects. In order to formalize this cardinality

constraint let us use o_P and o_V to denote the set of objects for the Profile and Video classes and r_{P-V} to denote the relation between Profile objects and Video objects. Then the constraint that corresponds to this relation is formalized as:

$$(\forall v \in o_V, \exists p \in o_P : (p, v) \in r_{P-V}) \wedge$$
$$(\forall p, p' \in o_P, \forall v \in o_V : ((p, v) \in r_{P-V} \wedge (p', v) \in r_{P-V}) \Rightarrow p = p') \quad (2.1)$$

Semantics of all of the data model declaration constructs we discussed above other than the dependency constraints can be formalized similarly [Nijjar and Bultan 2012].

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. Again, consider the relation between the Profile and Video objects. In order to model the delete operation, we have to specify the set of Profile objects, the set of Video objects and the relation between the Profile and Video objects both before and after the delete operation (o_P, o_V, r_{P-V}, o'_P, o'_V, r'_{P-V}, respectively). Then, to model the delete dependency expressed using the `:dependent` option in line 12 in Figure 2, we need to specify that when a Profile object is deleted, the Video objects related to that Profile are also deleted (and thus all the tuples, possibly containing other Profile objects, that are associated with those Video objects will also be deleted). Formally:

$$\exists p_d \in o_P : o'_P = o_P \setminus \{p_d\} \wedge$$
$$(\forall v : v \in o'_V \Leftrightarrow (v \in o_V \wedge (p_d, v) \notin o_{P-V})) \wedge$$
$$r'_{P-V} = r_{P-V} \cap (o'_P \times o'_V) \quad (2.2)$$

where p_d denotes the Profile object that is being deleted. Note that modeling of cyclic destroy dependencies would require the use of transitive closure. Our current framework does not handle cyclic destroy dependencies.

2.4. Data Model Verification Problem

Using the data model specification constructs we discussed above, a developer can specify complex relations among objects of an application. Since a typical application contains dozens of object classes with many relations among them, data model specifications can contain errors and omissions that can result in unexpected behaviors and bugs. Hence, it would be worthwhile to automatically verify the data models. Below, we formalize the data model verification problem by formally defining the data model instances and what it means for a data model instance to satisfy a given data model property.

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

Given a data model instance $\mathcal{I} = \langle O, R \rangle$, 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 four types of properties: 1) *state assertions* (denoted by A_S): these are properties that we expect to hold for each instance of the data model; 2) *behavior assertions* (denoted by A_B): these are properties that we expect to hold for each pair of instances that form a behavior of the data model; 3) *state predicates* (denoted by P_S): these are properties we expect to hold in some instance of the data model; and, finally, 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. We denote that a data model satisfies an assertion or a predicate as $\mathcal{M} \models A$ where:

$$\mathcal{M} \models A_S \Leftrightarrow \forall \mathcal{I} = \langle O, R \rangle, \mathcal{I} \models \mathcal{M} \Rightarrow R \models A_S \tag{2.3}$$

$$\mathcal{M} \models A_B \Leftrightarrow \forall \mathcal{I} = \langle O, R \rangle, \forall \mathcal{I}' = \langle O', R' \rangle, (\mathcal{I}, \mathcal{I}') \models \mathcal{M} \Rightarrow (R, R') \models A_B \tag{2.4}$$

$$\mathcal{M} \models P_S \Leftrightarrow \exists \mathcal{I} = \langle O, R \rangle, \mathcal{I} \models \mathcal{M} \Rightarrow R \models P_S \tag{2.5}$$

$$\mathcal{M} \models P_B \Leftrightarrow \exists \mathcal{I} = \langle O, R \rangle, \exists \mathcal{I}' = \langle O', R' \rangle, (\mathcal{I}, \mathcal{I}') \models \mathcal{M} \Rightarrow (R, R') \models P_B \tag{2.6}$$

The data model verification problem is, given a state or a behavior assertion or a state or a behavior predicate, determining whether the data model satisfies the given property.

3. PROPERTY SPECIFICATION AND INFERENCE

Most verification techniques and tools expect a set of properties as input. Verification process is only effective if the input properties are correctly and thoroughly specified. A verification tool cannot find an error in the input system if a property that exposes the error is not provided as input. Since manual specification of properties is time-consuming, error prone and lacks thoroughness, many errors can be missed during verification. Another disadvantage of the manual specification is that it requires familiarity with a formal modeling language, which is typically not the case for most developers.

To address these challenges and facilitate property specification, we present a set of property templates that we found to be useful and flexible enough to cover a wide variety of data model properties. These property templates are easy to use and do not require the developers to learn a formal modeling language. Moreover, we developed heuristics for automatic inference of several property templates. These heuristics look for certain patterns in the data model schema and automatically generate an instance of a corresponding property pattern if a match is found. We present the property templates and the automatic property inference heuristics below.

3.1. Property Templates

We identified seven property templates that characterize the most common properties we observed in our earlier research on data model verification [Nijjar and Bultan 2011; 2012]. These templates can easily be instantiated by the user for different classes and relations by providing the names of the relations as input.

We present the formal definitions of the seven property templates below. Of the seven property templates we list below, templates I and IV are state assertions, templates II and III are state predicates, and templates V, VI, and VII are behavior assertions. For the following, let $\mathcal{M}$ be the data model about which we are expressing the property. Let $\mathcal{I} = \langle O, R \rangle$, $\mathcal{I}' = \langle O', R' \rangle$ be data model instances, $r_{A-B}, r_{B-C}, r_{A-C} \in R$, $r'_{A-B}, r'_{B-C} \in R'$ and $o_A, o_B, o_C \in O$, $o'_A, o'_B \in O'$. Let $\mathcal{I} \models \mathcal{M}$ and $(\mathcal{I}, \mathcal{I}') \models \mathcal{M}$.

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

$$\textit{alwaysRelated}(r_{A-B}) \equiv \forall a \in o_A, \exists b \in o_B : (a, b) \in r_{A-B} \tag{3.1}$$

For example we can express the following property on the data model in Figure 2: *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,

$$\textit{multipleRelated}(r_{A-B}) \equiv \exists a \in o_A,\ \exists b_1, b_2 \in o_B : b_1 \neq b_2 \wedge (a, b_1) \in r_{A-B} \wedge (a, b_2) \in r_{A-B} \quad (3.2)$$

Note that, because this template is a state predicate, this formula is not required to hold for *every* instance of the model, but instead needs to hold on at least one instance of the data model. 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

$$\textit{someUnrelated}(r_{A-B}) \equiv \exists a \in o_A, \forall b \in o_B : (a, b) \notin r_{A-B} \quad (3.3)$$

For example, the property *someUnrelated*(User-Photo) means that it is possible to have a User without any Photos. This template is also a state predicate, meaning that this formula is expected to hold on at least one instance of the data model.

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

$$\begin{aligned}&\textit{transitive}(r_{A-B}, r_{B-C}, r_{A-C}) \equiv \\ &\qquad \forall a \in o_A, \forall c \in o_C : (a, c) \in r_{A-C} \Leftrightarrow \exists b \in o_B, (a, b) \in r_{A-B} \wedge (b, c) \in r_{B-C}\end{aligned} \quad (3.4)$$

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

V. *noOrphans* applies to situations where objects can potentially be orphaned. This occurs when there is a relation r_{A-B} between classes o_A and o_B and deletion of an element of class o_A results in an element of class o_B that is not related to any element of class o_A. This property template asserts that this scenario does not happen. Formally,

$$\textit{noOrphans}(r_{A-B}) \equiv \forall a \in o_A, \forall b' \in o'_B : a \notin o'_A \Rightarrow (\exists a' \in o'_A : (a', b') \in r'_{A-B}) \quad (3.5)$$

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. *deletePropagation* 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:

$$\textit{deletePropagation}(r_{A-B}) \equiv \forall a \in o_A, \forall b \in o_B : (a \notin o'_A \wedge (a, b) \in r_{A-B}) \Rightarrow b \notin o'_B \quad (3.6)$$

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

VII. *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,

$$\textit{noDeletePropagation}(r_{A-B}) \equiv \forall a \in o_A, \forall b \in o_B : (a \notin o'_A \wedge (a, b) \in r_{A-B}) \Rightarrow b \in o'_B \quad (3.7)$$

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

3.2. Property Inference

Below we present three heuristics for automatically inferring instances of three property templates defined in the previous section: *transitive*, *noOrphans* and *deletePropagation* These three property templates are the ones we encountered the most during our manual analysis of data models in our prior work [Nijjar and Bultan 2011; 2012].

The heuristics we present take as input the data model schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ and output a list of inferred properties. Each heuristic focuses on a sub-schema that contains only the relations that are relevant for the corresponding property type.

Note that the inferred properties may not necessarily hold on the model. Inference of properties that necessarily hold would be of no benefit to the developer as that would make only the *absence* of properties a useful signal. On the other hand, inferring properties too loosely would create a high ratio of false positives, wasting programmer time.

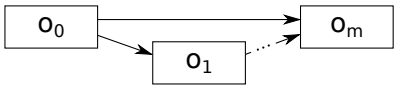

Fig. 6: The pattern used for inferring transitive relations.

3.2.1. Transitive Relations. The final property inference heuristic is for detecting transitive relations. 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 6. 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 1.

ALGORITHM 1: 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 ∈ 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 S' from o_1 to o_m then
            if p is of length 2 then
                Output transitive(r_0, r_1, r_2) ∨ transitive(r_2, r_1, r_0)
            else
                Output transitive(r_0, ..., r_m)
            end
        end
    end
end
```

3.2.2. 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

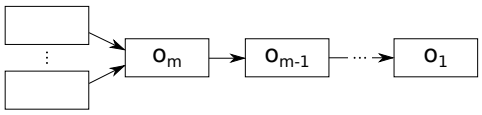

Fig. 7: The pattern used for recognizing orphan chains.

previous object class in the chain. Consider an object of the final class of a chain, such as o_{m-1} in Figure 7. 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.

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
  Output *noOrphans*(r_1)
end

3.2.3. 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. 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 8(a). Cycles in this graph are removed by collapsing strongly connected components to a single node. For the schema in Figure 8(a), nodes o_3 and o_4 are collapsed to a single node called c_1 in Figure 8(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 8(a) are given in Figure 8(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 *deletePropagation* 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 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 3.

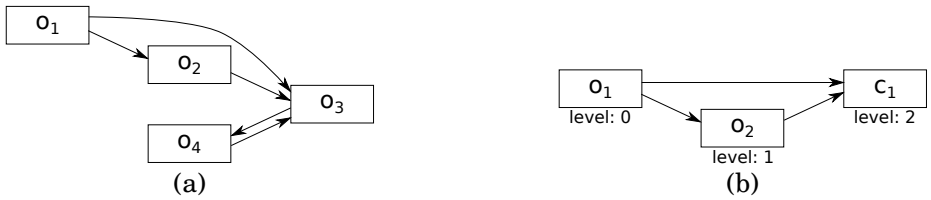

Fig. 8: A sub-schema (a) and the corresponding acyclic graph (b) constructed during the Inference Algorithm for Delete Propagation.

ALGORITHM 3: 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* 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
end
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 *deletePropagation*(r)
  end
end

4. VERIFICATION

We use automated verification techniques to verify the data model properties that are either specified using property templates or automatically inferred by the heuristics discussed in the previous section. We verify properties 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 two different variants of this framework: 1) a SAT-based bounded verification approach [Nijjar and Bultan 2011], and 2) a Satisfiability Modula Theories (SMT)-based unbounded verification approach [Nijjar and Bultan 2012].

The verification component of our tool is shown in Figure 9. It first converts the verification query to a logical formula and sends it to an SMT solver for unbounded verification. The unbounded verification phase can conclusively prove or disprove the property. If it disproves an assertion, it returns a counterexample data model instance demonstrating the failure of the assertion. Similarly, if it proves a predicate, it returns an instance demonstrating the satisfaction of a predicate. However, due to undecidability of unbounded verification, the SMT-based verification phase can produce an inconclusive result or a timeout. In this case, our tool converts the data model verification query to an Alloy specification and uses the Alloy Analyzer [Jackson 2006] for

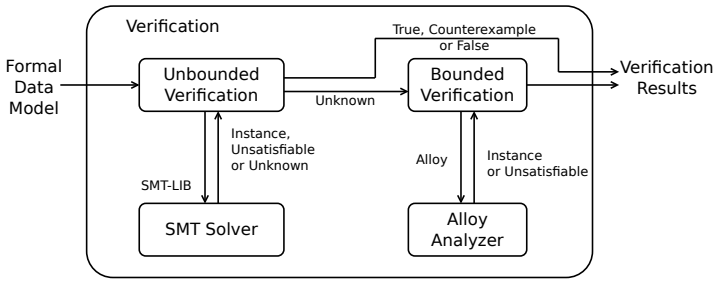

Fig. 9: Verification component of the toolset

SAT-based bounded verification. Using the Alloy Analyzer, it looks for a data model instance within a bound that violates the given assertion or satisfies the given predicate. If such an instance is found, it returns the data model instance. If such an instance is not found within the given bound, then the verification result can only be guaranteed to hold within that bound. Below, we discuss these two verification approaches.

4.1. Bounded Verification with Alloy

The main idea in bounded verification approach 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 if the following condition holds:

$$\exists \mathcal{I} = \langle O, R\rangle, \mathcal{I} \in \mathcal{I}_k \wedge \mathcal{I} \models \mathcal{M} \wedge R \not\models A_S \tag{4.1}$$

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 $\mathcal{I}_k$.

Similarly, given a state predicate P_S, we can check if the following condition holds:

$$\exists \mathcal{I} = \langle O, R\rangle, \mathcal{I} \in \mathcal{I}_k \wedge \mathcal{I} \models \mathcal{M} \wedge R \models P_S \tag{4.2}$$

In this case, if the condition holds, then we can conclude that the predicate P_S holds for the data model $\mathcal{M}$, i.e., $\mathcal{M} \models P_S$. However, if the condition does not hold, then we only know that the predicate P_S does not hold 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 [Jackson 2006] 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, our tool automatically translates ActiveRecord ORM specifications to Alloy specifications [Nijjar and Bultan 2011].

To demonstrate how the Alloy 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 our previous work [Nijjar and Bultan 2011]. After automatically translating the input data model and the property into an Alloy specification, our tool sends the specification to the Alloy Analyzer. Our tool then interprets the result returned by the Alloy Analyzer and reports back to the user whether the data model property failed or verified. It also returns a witness data model instance for assertions that fail and predicates that hold.

4.2. Unbounded Verification with an SMT Solver

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 ActiveRecord ORM code and a property, 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 [Nijjar and Bultan 2012]. Our tool translates the data model verification query into an SMT-LIB specification (which is a standard formula format used by SMT-solvers).

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

```
OBJECTS:
  User { User$0 }
  Tag { Tag$0 }
  Role { Role$0, Role$1, Role$2, Role$3, Role$4, Role$5, Role$6,
         Role$7, Role$8, Role$9, Role$10, Role$11, Role$12, Role$13 }

RELATIONS:
  --empty--
```

Fig. 10: A Data Store Instance Example

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. Details for the complete translation of the all data modeling constructs in Rails are provided in our previous work [Nijjar and Bultan 2012].

After translating the data model verification query into an SMT-LIB specification, our tool uses the SMT solver Z3 to determine the satisfiability of the generated formula. Based on the output of the SMT solver, it reports whether the property holds or fails. For assertions that fail and predicates that hold, it also reports a data model instance as a witness.

Since the SMT-based verification approach does not bound the sizes of the object classes or the relations, unlike the bounded verification case, if the verification tool reports that a property holds or fails, both results are conclusive. However, 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 [Bryant et al. 1999]. So, when the call to the SMT-solver times out or returns "unknown" we switch to the SAT-based bounded verification approach as shown in Figure 9.

4.3. Counter-Example Generation

If an assertion property fails, or if a predicate property is correct, we can produce an instance of the model that disproves the assertion or demonstrates the correctness of the predicate property.

Both Z3 and Alloy work by attempting to show that the input formulas are satisfiable. If they are satisfiable, a satisfying instance is generated. To prove predicate properties, we create the formulas corresponding to the model and the property and, if the translation is satisfiable, the predicate property holds. To prove assertion properties, we negate the property and ask Z3 or Alloy to show that this negated property is satisfiable within the model. If it is, there exists a data store instance that violates the property (the counter-example).

The formats in which these instances are given by Z3 and Alloy are difficult to interpret since they closely follow the syntactic details of the translation that our approach produces. To make these instances usable in practice, iDaVer automatically translates the instance into a more readable format. For example, if property `alwaysRelated[User, Profile]` were verified on the model in Figure 2, our method would show that this property does hold on the model and a satisfying instance that would be provided is presented in Figure 10. This format enumerates all objects of all classes that exist in the instance, and subsequently, how these objects are related. In this counterexample, there exists a User object (called `User$0`) and no objects are related (since the Relations section is empty).

5. REPAIR

Our tool automatically generates data model repairs for the failed properties and suggests them to the developer. These repairs show how the data model can be modified

```
1   class User < ActiveRecord::Base
2     has_and_belongs_to_many :roles
3     has_one :profile
4     has_many :photos
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
12    has_many :videos, :conditions => "format='mp4'"
13  end
14  class Photo < ActiveRecord::Base
15    belongs_to :user
16    belongs_to :profile
17    has_many :tags, :as => :taggable
18  end
19  class Video < ActiveRecord::Base
20    belongs_to :profile
21    has_many :tags, :as => :taggable
22  end
23  class Tag < ActiveRecord::Base
24    belongs_to :taggable, :polymorphic => true
25  end
```

(a) Before Repair

```
1   class User < ActiveRecord::Base
2     has_and_belongs_to_many :roles
3     has_one :profile, :dependent => :destroy
4     has_many :photos, :through => :profile
      validate :check_profile
      def check_profile
        if profile.nil?
          errors.add :profile, "Profile missing"
        end
      end
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, :conditions => "format='mp4'",
                        :dependent => :destroy
13  end
14  class Photo < ActiveRecord::Base
15    # line removed
16    belongs_to :profile
17    has_many :tags, :as => :taggable
18  end
19  class Video < ActiveRecord::Base
20    belongs_to :profile
21    has_many :tags, :as => :taggable
22  end
23  class Tag < ActiveRecord::Base
24    belongs_to :taggable, :polymorphic => true
25  end
```

(b) After Repair

Fig. 11: A Repair Example

(by changing the ORM source code) so that the failed property will hold in the repaired model.

A repair generated by our approach consists of a program point and the piece of code to be inserted at that program point. Each repair is either replacement of an association declaration (which augments an existing association declaration with some options) or insertion of runtime validation code.

The repair rules we developed for the property templates are discussed below. We will use the application given in Figure 11(a) as the running example in this section. This is a faulty version of the example given in Figure 2. The repaired version is given in Figure 11(b).

I. *alwaysRelated* fails if it is possible for an object to be unrelated to any object over a given relation $r = (o, t, n, o')$. Static cardinality constraints in ActiveRecord are not expressive enough to guarantee this property. Hence, this property must be enforced using runtime checks. Our automatically generated repair consists of runtime checks (validations) that enforce this property. These validations are run automatically by ActiveRecord whenever an object is about to be saved to the data store and will prevent invalid objects from reaching the data store.

For example, the property *alwaysRelated[User, Profile]* fails on the application in Figure 11(a). Our tool will generate the validation code between lines 4 and 5 in Figure 11(b) as the repair for this property.

II. *multipleRelated* fails if an object can be related to at most one object over a given relation $r = (o, t, n, o')$. To fix this property we generate a repair that alters the cardinality of the relation as follows: If the relation is `has_one` we make it `has_many`. If the relation is `belongs_to` with `has_many` on the other side, we make both sides of the relation

has_and_belongs_to_many. Otherwise, we have a belongs_to with has_one on the other side and this cannot be repaired directly as changing it to has_many and belongs_to would cause the side-effect of the other object having to be related to at least one of the first class. ActiveRecord cardinality constraints are not expressive enough to express repair in this case and in this case we generate runtime validations that enforce the property.

III. *someUnrelated* fails over a relation $r = (o, t, n, o')$ if it is impossible for an object of class o to be unrelated to another over r. As with the *alwaysRelated* property repair, this property is enforced by generating runtime checks.

IV. *transitive* 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 1) on the example in Figure 11(a) infers the following *transitive* property: the relation between User and Photo should be the composition of the relations between User and Profile, and Profile and Photo. However, we again find out that this property fails using automated verification. In other words, the photos in the profile associated with a user may not be the same as the photos associated with 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 Photo:

```
has_many :photos, :through => :profile
```

We also need to remove the belongs_to :user declaration in the Photo class since it becomes unnecessary when using the :through option. After this repair the relation between User and Photo will be the same as navigating the User-Profile relation and then the Profile-Photo relation. These repairs are presented in lines 4 and 15 of Figure 11(b).

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}.

V. *noOrphans* 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 11(a), a *noOrphans* property is generated which states that when a Profile is deleted no Video objects should be orphaned. This property fails when we check it using automated verification, which means that when a profile with videos is deleted, the videos are orphaned. In order to enforce this prop-

erty in the data model, a repair is generated that sets the `:dependent` on the relation with Videos in the Preference model, i.e.

```
has_many :videos, :conditions => "format='mp4'", :dependent => :destroy
```

This will cause the deletion of a Profile object to be propagated to the associated Videos. 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 11(a), is shown in Figure 11(b) between lines 12 and 13.

VI. *deletePropagation* If *deletePropagation*(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 3) on the data model of the application given in Figure 11(a), the *deletePropagation* property is generated for the relation between the User and Profile 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 profile of that user is not deleted. In order to enforce this in the data model, the repair our tool generates sets the `:dependent` option on the relation with Profile in the User model, i.e.

```
has_one :profile, :dependent => :destroy
```

as can be seen on line 3 of Figure 11. This will cause the deletion of User objects to be propagated to the associated Profile object. 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 repaired example (Figure 11), we observe that setting the `:dependent` option to `:delete` may result in Video objects with a dangling reference to deleted Profile objects. In order to prevent this inconsistency, the `:dependent` option is set to `:destroy` so that the Profile model can propagate the delete to the desired relations.

VII. *noDeletePropagation* is the dual of the *deletePropagation* property. As such, its repair is similar: `:destroy` and `:delete` dependencies are changed to `:nullify` option (which means that the deletion is not propagated further) in order to prevent deletion without causing dangling references.

6. EXPERIMENTS

To evaluate the effectiveness of the techniques, we ran our tool on seven open source Rails web applications. In choosing these applications, we focused on popular applications and attempted to cover a wide range of web application domains. Four of the seven applications we analyzed (FatFreeCRM, Lobsters, SprintApp, Tracks) are among the 25 most starred Rails applications on Github, according to the community maintained Ruby on Rails open source project index (www.opensourcerails.com). The remaining three applications (LovdByLess, OSR, Substruct) represent examples of a social network, an application repository, and a web store respectively. Since our implementation works only on Ruby version 1.8.7 and Rails version 2.x, the set of existing, open source applications that we can analyze was limited. With additional work, our toolset can be updated to handle the syntactic constructs added with the more recent versions.

We used our automatic inference algorithm to extract properties about the application, and in addition, used property templates to manually specify additional properties. These properties are sent to the next component of the tool which automatically translates the ActiveRecord files to SMT-LIB and performs verification using the

Z3 [Z3 2013] solver. If any properties time out during verification (with a time out limit of 1 minute), 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 (for example in the controller code). Properties enforced in other areas of the application can cause application errors in the future since if the application code is changed later on, 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 or data model properties that actually hold in the data model but our verification approach reports that they fail.

Table I: Sizes of the Applications

Application	Ruby LOC	Classes	Data Model Classes
FatFreeCRM	12069	54	20
Lobsters	4378	74	14
LovdByLess	3787	61	13
OSR	4295	41	15
SprintApp	3053	26	15
Substruct	15639	85	17
Tracks	6062	44	13

6.1. The Applications

The seven applications used in the experiments are listed in Table I, 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:

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

6.2. Inference and Verification Results

The results of running our tool on the seven applications are given in Table II. For each application and type of property, it displays the number of properties that were inferred by the tool, the number or properties for which unbounded verification timed out, the number of properties for which bounded verification did not produce a conclusive result within the bound, and the number of properties that were shown to

not hold during either bounded or unbounded verification. A total of 159 properties were inferred, of which 103 failed. There were no inconclusive results for the inferred properties. We manually specified a total of 43 properties using property templates, of which 14 are shown to not hold on the model, and 4 of which produced inconclusive verification results. All properties that timed out during unbounded verification were properties for FatFreeCRM and Lobsters applications, both of which have an exceptionally complex data model with a high number of polymorphic associations.

Table II: Inference and Verification Results per Application

Application	Property Source	Count	Timeout (SMT)	Inconclusive	Failed	Data Model and Application Errors	Data Model Errors	Failures Due to Rails Limitations	False Positives
FatFreeCRM	Inferred	25	9	0	12	0	8	1	3
	Templates	7	3	3	0	0	0	0	0
Lobsters	Inferred	21	19	0	19	7	3	0	9
	Templates	6	5	1	4	2	2	0	0
LovdByLess	Inferred	12	0	0	9	0	8	0	1
	Templates	8	0	0	2	0	2	0	0
OSR	Inferred	24	0	0	18	0	18	0	0
	Templates	3	0	0	3	0	3	0	0
SprintApp	Inferred	18	0	0	8	0	3	0	5
	Templates	5	0	0	0	0	0	0	0
Substruct	Inferred	31	0	0	18	0	5	5	8
	Templates	6	0	0	1	1	0	0	0
Tracks	Inferred	28	0	0	19	1	1	10	7
	Templates	8	0	0	4	1	3	0	0
Total	Inferred	159	28	0	103	8	46	16	33
	Templates	43	8	4	14	4	10	0	0

We manually investigated each of the failing properties to determine which correspond to data model errors. These results are also summarized in Table II.

For example, a *noOrphans* property that was inferred and failed verification (i.e., it 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 itself 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 in the data model without relying on the other parts of the application to establish 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 *deletePropagation* 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 *deletePropagation* 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 fails due to limitations in Rails' expressiveness.

As an example of a failure due to a different Rails limitation there is a *deletePropagation* 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 159 inferred properties for the seven web applications we analyzed, 56 properties (35.22%) hold on the given data model, 54 of them (33.96%) fail and correspond to data model errors (8 of which are also application errors), 16 of them (10.06%) fail due to Rails limitations, and 33 of them (20.75%) fail and correspond to false positives. Of the 43 properties we defined manually using templates, 29 properties (67.44%) hold on the given data model and 14 (32.56%) fail and correspond to data model errors (4 of which are also application errors). There were no failures due to Rails limitations and there were no false positives for the properties that were specified manually using templates. The fact that we are able to identify 12 application errors and an additional 56 data model errors across seven web applications indicates that data model errors are prevalent in web applications and web application developers are not using advanced features of ORMs effectively. We did not contact the developers directly about these bugs. 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.

Table III: Performance Information per Property Type

Inferred Property Type	#	Inference Time (ms)	Verification Time (ms)
transitive	34	103	40
noOrphans	9	10	27
deletePropagation	116	7	54

(a) Inferred properties

Template Property Type	#	Verification Time (s)
alwaysRelated	9	0.032
multipleRelated	4	0.051
someUnrelated	7	0.931
transitive	5	0.022
noOrphans	9	0.365
deletePropagation	5	1.783
noDeletePropagation	4	1.904

(b) Template properties

In Table III we summarize the performance of (a) inferred properties and (b) template properties across all seven applications. We observe that inferred properties can be verified very quickly, while template properties may take more time. Template properties occasionally timeout during unbounded verification and we included the

bounded verification time in this metric when it was necessary. We noticed that the complexity of the applications contributed more to the timeouts then the property itself, as all timeouts were on the most complex models (FatFreeCRM and Lobsters) and similar properties did not time out in simpler models. Bounded verification takes about 3 seconds while unbounded verification takes milliseconds and hence the disparity.

Table IV: Inference and Verification Performance per Application

Application	Property Source	Inference Time (ms)	Verification Time (s) SMT	Verification Time (s) Alloy	Formula Size (clauses) SMT	Formula Size (clauses) Alloy	Formula Size (variables) SMT	Formula Size (variables) Alloy
FatFreeCRM	Inferred	0.25	0.155	4.203	1363	265797	553	142670
	Templates	-	0.019	3.682	1246	263598	499	142152
Lobsters	Inferred	0.11	0.021	2.953	581	255488	257	120710
	Templates	-	0.015	2.944	477	258418	209	122163
LovdByLess	Inferred	0.09	0.037	-	546	-	245	-
	Templates	-	0.026	-	473	-	215	-
OSR	Inferred	0.13	0.029	-	501	-	221	-
	Templates	-	0.037	-	488	-	219	-
SprintApp	Inferred	0.16	0.027	-	475	-	228	-
	Templates	-	0.021	-	420	-	202	-
Substruct	Inferred	0.90	0.044	-	1051	-	486	-
	Templates	-	0.068	-	982	-	429	-
Tracks	Inferred	0.04	0.034	-	388	-	170	-
	Templates	-	0.031	-	344	-	154	-
Average	Inferred	0.28	0.049	3.355	713.9	258801	317.0	127768
	Templates	-	0.033	3.221	612.9	260360	271.1	129659

Table IV summarizes inference and verification performance averaged over each application and property generation method. For unbounded verification, the formula size measures the SMT-LIB specification produced for verification. 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 IV since it is close to zero for all properties.

Property inference was almost instantaneous in most cases, with transitive properties taking the longest time of about a millisecond. Unbounded verification usually took about 30 milliseconds, with the longest time being 155 milliseconds. Even bounded verification, which was necessary in 22% of cases, took 3 to 4 seconds, showing that our technique is applicable during development. In summary, our approach is not only able to find and repair errors effectively, it does so efficiently.

7. RELATED WORK

Formal modeling and automated verification of web applications have been investigated before. There has been some work on analyzing navigation behavior in web applications, focusing on correct handling of the control flow given the unique characteristics of web applications, such as the use of a browser's "back" button combined with the stateless nature of the underlying HTTP protocol [Krishnamurthi et al. 2006]. The problem of navigation inconsistencies in web applications has also been studied [Licata and Krishnamurthi 2004], where it has been shown that multiple browser windows can lead the user of a popular travel reservation site to purchase the wrong flight. Some language based solutions have been proposed to alleviate this problem, in which such navigation inconsistencies reduce to type checking errors [Krishnamurthi et al. 2006]. Prior work on formal modeling of web applications mainly focuses on state machine based formalisms to capture the navigation behavior. Modeling web applications as state machines was suggested a decade ago [Stotts et al. 1998] and investi-

gated further later on [Han and Hofmeister 2007; Book and Gruhn 2004; Hallé et al. 2010]. State machine based models have been used to automatically generate test sequences [Yuen et al. 2006], perform some form of model checking [Sciascio et al. 2005] and for runtime enforcement [Hallé et al. 2010]. In contrast to these previous efforts, we are focusing on analysis of the data model rather than the navigational aspects of a web application.

There has been some prior work on formal modeling of web applications using UML [Conallen 1999] and extending UML to capture complex web application behavior such as browsing and operations on navigation states [Baresi et al. 2001]. For example, WebML [Ceri et al. 2000] is a modeling language developed specifically for modeling web applications. There has also been recent work on specification and analysis of conceptual data models [Smaragdakis et al. 2009; McGill et al. 2011]. 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.

The idea of using patterns to facilitate formal property specification was first proposed for temporal logic properties [Dwyer et al. 1999]. The property templates we present in this paper are not temporal and they are specific to data model analysis.

Automated discovery of likely program invariants by observing runtime behaviors of programs has been studied extensively [Ernst et al. 1999; 2001; Ernst et al. 2007]. There has also been extensions of this style of analysis to inference of abstract data types [Guo et al. 2006]. Instead of using observations about the runtime behavior, we analyze the static structure of the data model extracted from ORM code to infer properties. Static verification of inferred properties has been investigated earlier [Nimmer and Ernst 2001]. 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 [Demsky and Rinard 2005; Demsky et al. 2006; Elkarablieh et al. 2007; Malik et al. 2009]. 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 code which is another distinguishing feature of our work.

There has been prior work on the verification of data models based on bounded verification using the Alloy Analyzer [Cunha and Pacheco 2009; Wang et al. 2006]. For example, mapping relational database schemas to Alloy has been studied before [Cunha and Pacheco 2009]. Also, translating ORA-SS specifications (a data modeling language for semi-structured data) to Alloy and using Alloy Analyzer to find an instance of the input data model has been investigated [Wang et al. 2006]. 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 [Bordbar and Anastasakis 2005] which is, a different class of errors than the ones we focus on in this paper.

Rubicon [Near and Jackson 2012] is a tool for verification of Controller (application logic) in Rails applications using Alloy Analyzer, whereas our work focuses on data model analysis. Further, we propose techniques to automatically infer properties, whereas Rubicon requires manual specifications. Finally, Rubicon, and all the other approaches mentioned above that use the Alloy Analyzer, are limited to bounded verification, whereas we perform both unbounded and bounded verification. There has been some other recent work on unbounded verification of Alloy specifications using SMT solvers [Ghazi and Taghdiri 2011], but to the best of our knowledge this approach has not been implemented.

ADSL [Bocic and Bultan 2014] is an approach on action verification in web applications. While the static portion of their model is very similar to the model presented in this paper, their work focuses on dynamic behavior and does not support property inference or template specification.

The discussion on inferring properties resembles the problem of detecting objects that are ready to be garbage collected [Wilson] (such as orphan chains, etc). However, the issue of garbage collecting is very different from the problem of avoiding orphans. In garbage-collecting memory management systems, garbage objects are the ones that can no longer be referred to (usually because the variable stack and the static portion of the runtime do not refer to the said object). This criterion cannot be applied to data stores because every data object can be queried and referred to. We refer to orphans on a semantic level, defined by the business logic.

8. CONCLUSIONS AND FUTURE WORK

In this paper we presented techniques for property specification, 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 present a set of property patterns that can be used to specify properties about the data model. We also present techniques that analyze the structure of the relations in the data model schema to automatically infer properties. Next we use automated verification techniques to check if the specified or 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.

The approach we present in this paper extracts a static data model from the association declarations and their options in ORM code. As future work we plan to extend this static approach by modeling the dynamic behavior, which we plan to extract by analyzing the methods that update the data model. Such an extension would enable us to find errors in method implementations which are not visible in the static data model analyzed in this paper.

REFERENCES

Luciano Baresi, Franca Garzotto, and Paolo Paolini. 2001. Extending UML for Modeling Web Applications. In *Proc. 34th Ann. Hawaii Int. Conf. Sys. Sci. (HICSS)*.

Ivan Bocic and Tevfik Bultan. 2014. Inductive verification of data model invariants for web applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 620–631.

Matthias Book and Volker Gruhn. 2004. Modeling Web-Based Dialog Flows for Automatic Dialog Control. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*. 100–109.

Behzad Bordbar and Kyriakos Anastasakis. 2005. MDA and Analysis of Web Applications. In *Proceedings of the VLDB Workshop on Trends in Enterprise Application Architecture (TEAA)*. 44–55.

Randal E. Bryant, Steven M. German, and Miroslav N. Velev. 1999. Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*. 470–482.

Stefano Ceri, Piero Fraternali, and Aldo Bongio. 2000. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks* 33, 1-6 (2000), 137–157.

Jim Conallen. 1999. Modeling Web Application Architectures with UML. *Commun. ACM* 42, 10 (1999), 63–70.

Alcino Cunha and Hugo Pacheco. 2009. Mapping between Alloy Specifications and Database Implementations. In *Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. 285–294.

Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. 2006. Inference and enforcement of data structure consistency specifications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 233–244.

Brian Demsky and Martin C. Rinard. 2005. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. 176–185.

M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering*. 411–420.

Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. 2007. Assertion-based repair of complex data structures. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 64–73.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*. 213–224.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1-3 (2007), 35–45.

Aboubakr Achraf El Ghazi and Mana Taghdiri. 2011. Relational Reasoning via SMT Solving. In *Proceedings of the 17th International Symposium on Formal Methods (FM)*. 133–148.

Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. 2006. Dynamic inference of abstract types. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 255–265.

Sylvain Hallé, Taylor Ettema, Chris Bunch, and Tevfik Bultan. 2010. 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)*. 235–244.

Minmin Han and Christine Hofmeister. 2007. Relating Navigation and Request Routing Models in Web Applications. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. 346–359.

Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Massachusetts.

Glenn E. Krasner and Stephen T. Pope. 1988. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Jour. Object-Orient. Program.* 1, 3 (1988), 26–49.

Shriram Krishnamurthi, Robert Bruce Findler, Paul Graunke, and Matthias Felleisen. 2006. *Modeling Web Interactions and Errors*. Springer, 255–275.

Daniel R. Licata and Shriram Krishnamurthi. 2004. Verifying Interactive Web Programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*. 164–173.

Muhammad Zubair Malik, Khalid Ghori, Bassem Elkarablieh, and Sarfraz Khurshid. 2009. A Case for Automated Debugging Using Data Structure Repair. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 620–624.

Matthew J. McGill, Laura K. Dillon, and R. E. Kurt Stirewalt. 2011. Scalable analysis of conceptual data models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 56–66.

Joseph P. Near and Daniel Jackson. 2012. Rubicon: Bounded verification of web applications. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. 60.

Jaideep Nijjar, Ivan Bocic, and Tevfik Bultan. 2013. An Integrated Data Model Verifier with Property Templates. In *Proc. FormaliSE*. IEEE, 29–35.

Jaideep Nijjar and Tevfik Bultan. 2011. Bounded verification of Ruby on Rails data models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 67–77.

Jaideep Nijjar and Tevfik Bultan. 2012. Unbounded Data Model Verification Using SMT Solvers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 210–219.

Jaideep Nijjar and Tevfik Bultan. 2013. Data model property inference and repair. In *ISSTA*, Mauro Pezzè and Mark Harman (Eds.). ACM, 202–212.

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

Eugenio Di Sciascio, Francesco M. Donini, Marina Mongiello, Rodolfo Totaro, and Daniela Castelluccia. 2005. Design Verification of Web Applications Using Symbolic Model Checking. In *Proc. 5th Int. Conf. Web Engineering (ICWE)*. 69–74.

Yannis Smaragdakis, Christoph Csallner, and Ranjith Subramanian. 2007. Scalable automatic test data generation from modeling diagrams. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. 4–13.

Yannis Smaragdakis, Christoph Csallner, and Ranjith Subramanian. 2009. Scalable satisfiability checking and test data generation from modeling diagrams. *Automated Software Engineering* 16, 1 (2009), 73–99.

P. David Stotts, Richard Furuta, and Cyrano Ruiz Cabarrus. 1998. Hyperdocuments as Automata: Verification of Trace-Based Browsing Properties by Model Checking. *ACM Trans. Inf. Syst.* 16, 1 (1998), 1–30.

Lin Wang, Gillian Dobbie, Jing Sun, and Lindsay Groves. 2006. Validating ORA-SS Data Models using Alloy. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*. 231–242.

Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management, (IWMM 92)*.

Shoji Yuen, Keishi Kato, Daiju Kato, , and Kiyoshi Agusa. 2006. Web Automata: A Behavioral Model of Web applications based on the MVC model. *Information and Media Technologies* 1, 1 (2006), 66–79.

Z3 2013. Z3. http://research.microsoft.com/projects/z3/. (2013).

Received January 2014; revised ?; accepted ?