UNIVERSITY OF CALIFORNIA
Santa Barbara

Analysis and Verification of Web Application Data Models

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jaideep Nijjar

Committee in Charge:

Professor Tevfik Bultan, Chair

Professor Chandra Krintz

Professor Jianwen Su

March 2014

The Dissertation of
Jaideep Nijjar is approved:

Professor Chandra Krintz

Professor Jianwen Su

Professor Tevfik Bultan, Committee Chairperson

March 2014

To my parents,

Gurmit Singh Njjar and Rupinderjit Kaur Nijjar.

Acknowledgements

I would like to thank my family and friends for all their love and support over the years.

I am grateful for my colleagues, Ivan Bocic and Muath Alkhalaf. I had many enlightening conversations with Bo, which directly impacted the quality of my research. Muath, being a senior researcher, felt like a guide and was always generous with his advice. His perspectives and experiences as a grad student provided comfort and insight.

I acknowledge the WiCS group (Women in Compute Science) for the kind of support only fellow women can provide. Thank you for organizing non-school related events, for the chance to connect and talk outside of the classroom and research lab, and for the opportunity to give back to the group during my term as Treasurer.

Thank you to the entire CS department at UCSB. It has been an honor to be part of such a talented, caring and supportive group of people. The financial, emotional and academic support provided by this department has been key in my success as a graduate student.

Deep gratitude to Jianwen Su and Chandra Krintz, members of my Ph.D. committee. I thank them for all the feedback, criticism, and suggestions they have offered. The quality of the work presented in this dissertation has improved greatly due to them.

I especially thank my research advisor, Tevfik Bultan. Tevfik Bultan's dedication to his students is outstanding. He has a gift in transforming students into independent researchers and leaders in their field. He does it very gently, holding your hand as you enter the unfamiliar and tough research world, and lets go exactly when the time is right, confident in your abilities even when you are not.

I thank him for the positive feedback and the constructive criticism he has given, and for the time he takes to give detailed corrections and suggestions. I appreciate him asking his students for their feedback of their own work, progress and experience. And then for offering the support and flexibility to develop our own research style, thereby bringing out the best in us.

Tevfik Bultan takes pride in doing excellent work, and inspires his students to do the same. He is knowledgeable and phenomenal in all aspects of the his position, which includes teaching, writing papers, presenting, grant-writing and mentoring. Further, he exhibits admirable work-life balance; he teaches his students to be well-rounded by example.

Thank you, professor Tevfik Bultan, for your support, encouragement and guidance all these years. Considering the number of times I was about to drop out of the program, this Ph.D. would not have been possible without your honest and supportive words. It has been an honor working with someone with such determination, passion, integrity, compassion, knowledge, and patience. Overall, you have been an incredible source of inspiration. Thank you.

Curriculum Vitæ

Jaideep Nijjar

Education

2013	Master of Science in Computer Science, University of California, Santa Barbara.
2008	Bachelor of Science in Computer Science, Seattle University.
2008	Bachelor of Science in Mathematics, Seattle University.

Publications

Jaideep Nijjar, Ivan Bocic, and Tevfik Bultan. "Data Model Property Inference, Verification and Repair for Web Applications." Submitted to *ACM Transactions on Software Engineering and Methodology.*

Jaideep Nijjar, Ivan Bocic, and Tevfik Bultan. "An Integrated Data Model Verifier with Property Templates." In Proceedings of the *FME Workshop on Formal Methods in Software Engineering (FormaliSE),* pages 29-35, 2013.

Jaideep Nijjar and Tevfik Bultan. "Data model Property Inference and Repair." In Proceedings of the *International Symposium on Software Testing and Analysis (ISSTA),* pages 202-212, 2013.

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

Jaideep Nijjar and Tevfik Bultan. "Bounded Verification of Ruby on Rails Data Models." In Proceedings of the *International Symposium on Software Testing and Analysis (ISSTA),* pages 67-77, 2011.

Abstract

Analysis and Verification of Web Application Data Models

Jaideep Nijjar

Nowadays many software applications are deployed over compute clouds using the three-tier architecture, where the persistent data for the application is stored in a backend datastore and is accessed and modified by the server-side code based on the user interactions at the client-side. The data model forms the foundation of these three tiers, and identifies the set of objects stored by the application and the relations (associations) among them. In this dissertation, we present automated techniques for data model specification, verification and repair.

We first present an approach for automated verification of data models that 1) extracts a formal data model from an object-relational mapping, 2) converts verification queries about the data model to queries about the satisfiability of logic formulas, and 3) uses an automated decision procedure to check the satisfiability of the resulting formulas. To improve the ease-of-use for this formal verification framework, we provide property templates for specifying data model properties, which are automatically translated to logical formulas before verification.

To further automate the specification and verification process, we present techniques for automatically inferring properties about the data model by analyzing the relations among the object classes, and identifying patterns that correspond to a subset of our property templates. We then check the inferred properties on the data model using our automated verification techniques. For the properties that fail, we present techniques that generate fixes to the data model that establish the inferred properties.

We implemented these techniques in a tool for analyzing web applications built using the Ruby on Rails framework and applied it to five open source applications. Our experimental results demonstrate that our approach is effective in automatically identifying and fixing errors in data models of real-world web applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although the web began as a medium for sharing information stored in static HTML pages, it has evolved into an ubiquitous medium for interaction and dynamism, driven by sophisticated web applications that can provide highly complex functionality that was once only available in desktop applications. Early on, e-commerce was the driving force behind the development of complex browser-based web applications; more recently, social networking has assumed this role. We are now entering a new phase of the web revolution where web applications are replacing desktop applications. Software-as-a-service supported with cloud computing platforms has become a realistic and attractive alternative to stand-alone desktop applications, saving users from the constant hassles of installation, configuration, software upgrades and security patches. The use of web applications has increased rapidly with the expanding growth of mobile phones, tablets and e-readers. This is certainly not the last phase of the web revolution. In the near future, web applications will play a significant role in improving the efficiency of national infrastructures in many critical areas such as healthcare [40], national security, and the power grid [41].

There is a large stumbling block to this ever-increasing reliance on web applications: web applications are not dependable. For example, they are notorious for security vulnerabilities [79] and are easily confused by unexpected user requests [42]. Given their increasing importance and ubiquity, the lack of dependability in web applications can no longer be tolerated. There is an urgent need for developing techniques and tools that can be used to improve the dependability of web applications. To be successful, these techniques and tools must take into account and exploit the unique characteristics of web applications and their common architectural patterns.

It has become common practice to write web applications using scripting languages, such as Ruby, because of their quick turnaround times in producing working applications. However, because of their dynamic nature, it is easy to introduce hard-to-find bugs in the applications written using these scripting languages. Current web software development processes rely on manual testing for eliminating bugs. Although testing is necessary for improving the dependability of software systems in general, it is not possible to cover the state space of a web application using testing. Hence, undetected bugs find their way into deployed software systems resulting in unreliable behavior at best, and critical safety and security flaws at worst.

This is where static verification comes in. Static verification allows a rigorous method of analyzing applications systematically and exhaustively. Another benefit of using static verification is that it can be applied during the earlier phases of software development. Testing is usually the last phase of software development and must be executed after implementation is complete. However, it is widely known that the majority of the cost of software development and maintenance comes from design defects. Thus, the methodology used to identify such defects should be one that takes place earlier in the software development cycle. Verification based on static analysis is one such methodology. By catching errors in the design phase of web application development, it helps decrease the cost of maintenance, debugging and patching flawed designs.

Most modern software applications are developed using the three-tier architecture (Figure 1.1) that consists of a client, a server and a backend datastore. The client-side code is responsible for coordinating the interaction with the user. The server-side code implements the business logic and determines the control flow of the application. The backend datastore stores the persistent data for

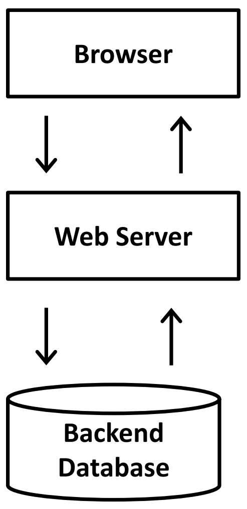


Figure 1.1: The Three-Tier Architecture

the application. The interaction between the server and the backend datastore is typically managed using an object-relational mapping (ORM) that maps the object-oriented code at the server side to the relational database at the backend. The ORM makes use of a data model to accomplish this.

A data model specifies the types of objects (*e.g.*, user, account, etc.) stored by the application and the relations among the objects (*e.g.*, the relation between users and accounts). A data model also specifies constraints on the data model relations (*e.g.*, the relation between two object types must be one-to-one). The ORM maps this data model to the appropriate representations used by the server-side code and the backend datastore. Since data models form the foundation of such applications, their correctness is of paramount importance for the dependability of such applications.

The focus of our work is the verification of data models in web applications. Web applications have been the most successful type of cloud-based software applications. There are many frameworks that support their development, but recently frameworks employing the Model-View-Controller (MVC) pattern [55] have gained popularity. Ruby on Rails (Rails for short), Zend for PHP, CakePHP, Django for Python, and Spring for J2EE are all examples of frameworks based on the MVC pattern. This pattern facilitates the separation of the data model (Model) from the user interface logic (View) and the control flow logic (Controller). (See Fig-

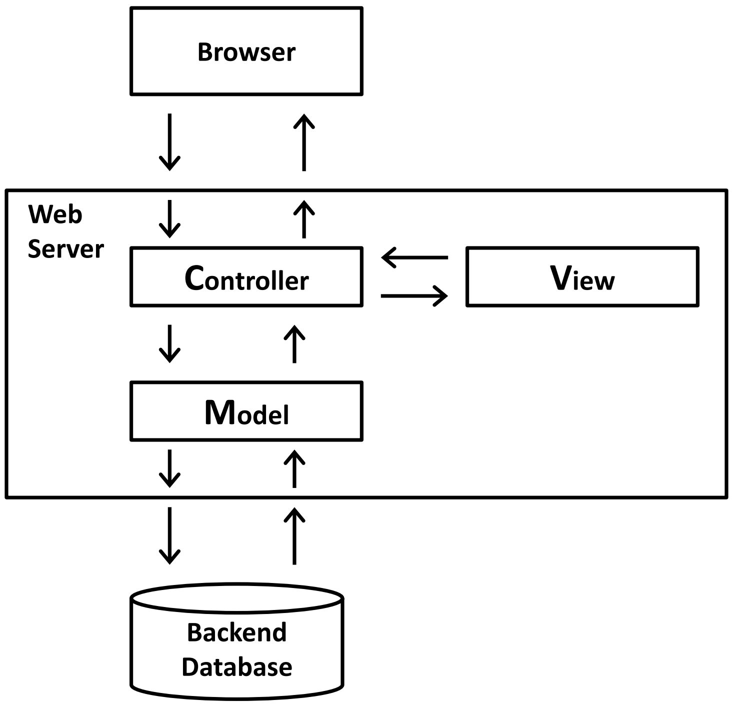


Figure 1.2: The Model-View-Controller Pattern

ure 1.2.) 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.

In this dissertation we present an automated verification approach for data models. It includes bounded verification of data models using Alloy as well as unbounded verification using the theory of uninterpreted functions with quantification. These two approaches were integrated and implemented as tool for the Ruby on Rails framework (Figure 1.3). The front-end of our tool, iDaVer, automatically extracts a formal data model from the ORM specification of the input Rails application. The user writes properties about the data model that she expects to hold, and then chooses to perform either bounded or unbounded verification.

Although the formal data model is extracted automatically, the user still has to specify the properties she desires to check about the data model. To facilitate this process, we developed a set of property templates. These templates characterize the most common properties we observed in our research on data model verification [72, 73]. These templates can easily be instantiated for different classes and relations by the user.

In order to verify properties (specified using property templates) using bounded verification, the tool first automatically translates the Rails data models to formal specifications in the Alloy language [54]. Next, we use the bounded verification techniques implemented in the Alloy Analyzer to check if the specified properties hold on all instances of the given data model within a given bound. Alloy Analyzer converts bounded verification queries to Boolean SAT problems and uses a SAT-solver to determine the result. The exhaustive exploration either finds a violation of a given property (which indicates an error in the data model) or it

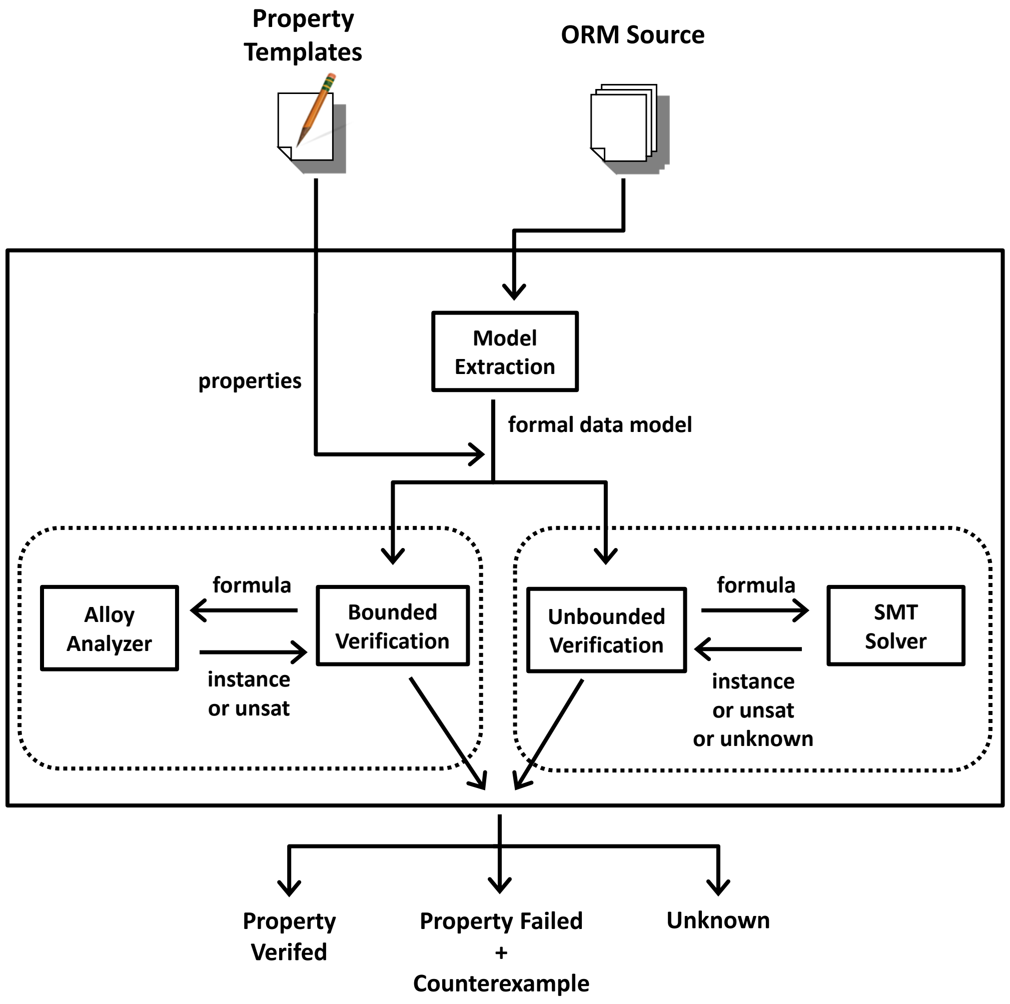


Figure 1.3: IDAVER's Architecture

terminates without finding a violation (which guarantees that the data model does not violate the specified properties within the given bound).

If the user would like to perform unbounded verification, iDaVer converts verification queries to formulas in the theory of uninterpreted functions and then uses a Satisfiability Modulo Theories (SMT) solver to determine the satisfiability of the queries. Based on the satisfiability result of the generated SMT formula, our tool reports whether the property holds on the data model, fails to hold, or that the SMT solver timed out during verification. Timeouts may occur due to the undecidability of the theory of uninterpreted functions with quantification [11]. Our SMT-based verification approach does not bound the sizes of the object classes or the relations, so if our verification tool reports that an assertion holds, it is guaranteed to hold for any data model instance. In both the bounded and unbounded approaches, our tool provides a sample data model as a counterexample for failing assertions.

To evaluate the effectiveness and usability of our approach, we applied it to five open-source Rails applications and identified a variety of data model errors. Our results show that our verification techniques are feasible and useful in analyzing real-world applications. We include a performance comparison of the bounded and unbounded approaches.

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

Once the automatically inferred properties are generated, a data model verification technique (such as the ones we developed and introduced earlier) can be used to determine if the inferred properties are actually enforced by the data model.

Our techniques are applicable to ORMs in general, and we have implemented the approach for the Ruby on Rails framework. The techniques have been integrated as a complete toolset for data model analysis, verification and repair. We used our toolset to analyze five open source Ruby on Rails applications. Our results indicate that the integrated automated property inference, verification and repair approach is effective in discovering and eliminating errors in data models of real-world web applications.

In summary, the novel contributions of this dissertation include:

1. A data model verification technique that automatically translates data model verification queries to formulas in the Alloy language and uses a boolean SAT solver to answer them.

2. A data model verification technique that can handle unbounded data models by automatically translating verification queries to formulas in the theory of uninterpreted functions and using SMT solvers to answer them.

3. iDaVer, an integrated tool that combines SAT- and SMT-based data model verification approaches with property templates.

4. Automated techniques for property inference that extract a data model schema from the ORM, investigate the structure of the generated data model schema and generate properties that are expected to hold in the data model.

5. The integration of automated property inference techniques with automated verification techniques in order to identify which of the inferred properties are valid based on the semantics of the data model.

6. Automated repair generation techniques that propose modifications to the data model so that the modified data model satisfies the properties that fail.

7. Implementation of the proposed approaches for the Ruby on Rails framework.

8. Experimental evaluation of the proposed approaches on five real-world applications.

The remainder of this dissertation is organized as follows: Chapter 2 describes data models in web applications, focusing on the Ruby on Rails framework. Chapter 3 gives a formal description of the data model verification problem. Chapter 4 describes our bounded verification approach and how we translate Rails data models to the Alloy language. Chapter 5 presents our unbounded verification approach and the translation of data model verification queries to the theory of uninterpreted functions. Chapter 6 presents our data model verification experiments. Chapter 7 describes our data model property inference algorithms, our automated repair generation techniques, and related experiments. Chapter 8 presents data model property templates and iDaVer, our tool for integrated data model verification. Related work is discussed in Chapter 9 and we conclude in Chapter 10.

Chapter 2

Data Models in Web Applications

The data model forms the foundation of a web application that uses the three-tier architecture. Any error in this foundation can have a significant impact on the entire application. The data model provides an abstraction between the application code and the backend datastore in a web application. Typically, from the backend datastore's point of view the data is stored in a relational database, whereas from the perspective of the application code the data is represented using an object-oriented data model. The object-relational mapping (ORM) used by the three-tier architecture handles the translation between these two views, so that the application code and the backend datastore can interact with each other while preserving their own views of the data. In this chapter we describe how data models are expressed using the Ruby on Rails framework as an example.

2.1 Rails Data Models and the Basic Relationships

The ORM used by Ruby on Rails is called Active Records. Active Records handle all the details of connecting to the underlying database, mapping objects to tables, and data manipulation. Active Records are also used to manage relationships between tables.

We will use the running example shown in Figure 2.1 to describe the various data-modeling features available in Rails. Figure 2.1 displays a simple Active Records specification for a social networking application where users have profiles which store their photo and video files. The photos and videos can be tagged by users, and users can have different roles.

Active Records handle three basic types of relationships:

1. *one-to-one:* An ObjectA is associated with zero or one ObjectB's. So, more accurately, this is a one-to-zero-or-one relationship.

2. *one-to-many:* An ObjectA is associated with an arbitrary (zero or more) number of ObjectB's.

3. *many-to-many:* An arbitrary (zero or more) number of ObjectA's are associated with an arbitrary (zero or more) number of ObjectB's.

```ruby
class User < ActiveRecord::Base
        has_and_belongs_to_many :roles
        has_one :profile, :dependent => :destroy
        has_many :photos, :through => :profile
end
class Role < ActiveRecord::Base
        has_and_belongs_to_many :users
end
class Profile < ActiveRecord::Base
        belongs_to :user
        has_many :photos, :dependent => :destroy
        has_many :videos, :dependent => :destroy,
                          :conditions => "format='mp4'"
end
class Photo < ActiveRecord::Base
        belongs_to :profile
        has_many :tags, :as => :taggable
end
class Video < ActiveRecord::Base
        belongs_to :profile
        has_many :tags, :as => :taggable
end
class Tag < ActiveRecord::Base
        belongs_to :taggable, :polymorphic => true
end
```

Figure 2.1: A data model example

These relationships are expressed by adding a pair of declarations (from the set of four declarations: `has_one`, `has_many`, `belongs_to`, and `has_and_belongs_to_many`) in the corresponding Active Record file of the related objects. The pair of declaration required for each relation type is explained below using the running example.

1. one-to-one: To declare a one-to-one relationship between User and Profile objects, the `has_one` and `belongs_to` declaration pair is used:

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

2. one-to-many: To declare a one-to-many relationship between Profile and Photo objects, the `has_many` and `belongs_to` declaration pair is used:

   ```
   class Profile < ActiveRecord::Base
     has_many :photos
   end
   class Photo < ActiveRecord::Base
     belongs_to :profile
   end
   ```

3. many-to-many: To declare a many-to-many relationship between User and Role objects the `has_and_belongs_to_many` declaration is used in both Active Record files:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :roles
end
class Role < ActiveRecord::Base
  has_and_belongs_to_many :users
end
```

Note that in order to express a relationship that an object has with itself, one would put both declarations in the same class. Also note that the < operator is used to express inheritance in Rails. All the objects we have discussed so far inherit from the `ActiveRecord::Base` class. This is so that the data objects inherit all the database-connection functionality that is located in the `ActiveRecord::Base` class. Classes can also inherit from user-declared classes.

2.2 Extending the Basic Relationships using Options

Rails provides a set of options that can be used to extend the three basic relationships we discussed above. We discuss the four options that affect relationships between data objects, below.

The `:through` Option The first is the `:through` option, which can be set on the `has_many` and `has_one` declarations. One use of the `:through` option is for setting up a join table for a many-to-many relation, as opposed to a join model using the `has_and_belongs_to_many` declaration. Another use of the `:through` option is when ObjectA has a one-to-many or one-to-one relation with ObjectB, ObjectC also has a one-to-many or one-to-one relation with ObjectB, and the Rails programmer would like direct access from ObjectA to ObjectC. For instance (see below), in the running example User has a one-to-one relation with Profile and Profile has a one-to-many relation with Photo. To get all the photos associated with a profile that belongs to a user, a programmer would typically have to write code to get all the Profile object associated with the User object, and then write code to obtain the set of Photo objects associated with that Profile. Instead, by using the `:through` option, the programmer can declare that Users have many Photos

`:through` Profile. This will allow the programmer to directly access the set of Photos from a User object.

```
class User < ActiveRecord::Base
  has_one :profile
  has_many :photos, :through => :profile
end
class Profile < ActiveRecord::Base
  belongs_to :user
  has_many :photos
end
class Photo < ActiveRecord::Base
  belongs_to :profile
end
```

The `:conditions` Option The second option for extending relationships is the `:conditions` option, which can be set on all of the four declarations (`has_one`, `has_many`, `belongs_to`, and `has_and_belongs_to_many`). As an example of its use, consider the following:

```
class Profile < ActiveRecord::Base
  has_many :videos, :conditions => "format='mp4'"
end
class Video < ActiveRecord::Base
  belongs_to :profile
end
```

The `:conditions` option limits the relationship to those objects that meet a certain criteria. In this example, Profile objects are only related to a Video object if its `format` field is `'mp4'`. The condition statement needs to be in the form of the WHERE clause of a SQL query.

The `:polymorphic` Option The third option for extending relationships is the `:polymorphic` option to the `belongs_to` declaration, which Rails uses to declare polymorphic associations. This is similar to the idea of interfaces in object-oriented design, where dissimilar objects have common characteristics that are embodied in the interface they implement. In the running example this occurs with Photos and Videos. Although Photos and Videos are not similar enough to have a sub-class relationship, they both can have Tags. By using the `:polymorphic` option in the Tag class we can allow any and all classes that have tags to create an association with the Tag class. Any classes created in the future can also take part in this relationship, all without having to make any changes to the Tag class. The models for the Photo, Video and Tag classes, with the polymorphism constructs highlighted, are given below.

```
class Tag < ActiveRecord::Base
  belongs_to :taggable, :polymorphic => true
end
class Photo < ActiveRecord::Base
  has_many :tags, :as => :taggable
end
class Video < ActiveRecord::Base
  has_many :tags, :as => :taggable
end
```

The polymorphic relationship is expressed in Photo and Video using the `has_many` declaration with the `:as` option. The `:as` option can also be specified on a `has_one` declaration. Next, the Tag class requires a `:belongs_to` declaration with the `:polymorphic` option set. This allows a `Tag` to belong to any taggable classes.

The `:dependent` Option The final Rails construct we discuss adds some dynamism to the data model; it allows modeling of object deletion at the data model level. The Rails construct for this is the `:dependent` option, which can be set for all the relation declarations except `:has_and_belongs_to_many`. Normally when an object is deleted, its related objects are not deleted. However, by setting the `:dependent` option to `:destroy` or `:delete` (`:delete_all` for the `has_many` declaration), deleting this object will also delete any associated object(s). Although there are several differences between `:destroy` and `:delete`, the one that is important for our purposes is that `:delete` will directly delete the associated object(s) from the database without looking at its dependencies, whereas `:destroy` first checks whether the associated objects themselves have relations with the `:dependent` option set. For an example of the use of the `:dependent` option, consider the following excerpt from data model of the running example:

```
class User < ActiveRecord::Base
  has_one :profile, :dependent => :destroy
end
```

The relation in the User class with Profile has the `:dependent` option set. This means that when a User object is deleted, the profile object related to it is also deleted. Further, since the `:dependent` option is set to `:destroy`, any relations with the `:dependent` option set in the Profile class will also have their objects deleted. In the case of the running example, the Profile class has two relations with the `:dependent` option set (`photos` and `videos`) to which the delete will propagate:

```
class Profile < ActiveRecord::Base
  belongs_to :user
  has_many :photos, :dependent => :destroy
  has_many :videos, :dependent => :destroy
end
```

2.3 Data Modeling Constructs in Other Object-Relational Mappings

The constructs presented so far form the essence of Rails data models specified using Active Record. The basic data modeling constructs are also supported by other object-relational mappings (ORMs) such as Hibernate and DjangoORM. In order to demonstrate that the basic types of relations can also be specified in other ORMs, we provide a simple syntax comparison between these three different ORM libraries (Active Record, Hibernate, and DjangoORM) in Figure 2.2, where objects of class Foo are associated with objects of class Bar with different cardinality constraints [70].

Active Record is the most expressive among these ORM libraries, and allows declaration of complex relations using the options we discussed above. However, many features of Active Record are available in other ORMs as well. For example, Hibernate supports delete propagation in a manner similar to `:dependent => :destroy` option of Active Record. 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 Active Record is unavailable in Hibernate or DjangoORM (though easy to implement manually), and DjangoORM does not have a feature similar to Active Record's `:dependent` option. Instead, delete propagation has to be implemented manually.

Using constructs like those discussed in this chapter, a developer can specify complex relations among objects of an application. Since a typical application would contain dozens of object classes with many relations among them, it is possible to have errors and omissions in the data model specification that can result in unexpected behaviors and bugs. It is our goal in this dissertation is to present an automated verification tool that can help analyze a data model and identify errors.

Cardinality	Active Record	Hibernate	DjangoORM

One to Zero or One	`class Foo`
	`  has_one :bar`
	`end`
	
	`class Bar`
	`  belongs_to :foo`
	`end`
		`public class Foo {`
		`  @OneToOne(mapped_by = "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 :bars`
	`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 :bars`
	`end`
	
	`class Bar`
	`  has_and_belongs_to_many :foos`
	`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)`

Figure 2.2: ORM Library Syntax Comparison

Chapter 3

Formalizing Data Models

We are now ready to discuss how data model constructs can be formalized as relational constraints on a formal data model. A formal data model represents the objects and their relationships in an application. In this chapter we will explain how data models in web applications can be formalized, using Active Records as an example. Note that we focus on analyzing a mainly static model. We do not model operations that update objects and their relationships, except for delete propagation which is declared in Active Records using the `:dependent` option. By focusing on the static data model specified in the Active Record files we can extract the set of constraints that must hold for any instance of the data model. In this chapter we present the formal data model and formalize the data model verification problem.

3.1 The Formal Data Model

We define a data model as a tuple $M = \langle S, C, D \rangle$ where S is the data model schema, identifying the sets and relations of the data model, C is a set of relational constraints and D is a set of dependency constraints. The schema S only identifies the names of the object classes, the names of the relations and the domains and ranges of the relations in the data model. For example, the schema for the example shown in Figure 2.1 will identify the following set of object classes {User, Role, Profile, Photo, Video, Tag} and the relations among these object classes {photo-profile, photo-tag, photo-user, profile-user, profile-video, role-user, tag-video}, where each relation has an identified domain and range (we named the relations above so that the prefix identifies the domain and the suffix identifies the range).

The relational constraints in C express all the constraints on the relations such as the ones related to cardinality (one-to-one, one-to-many, and many-to-many), the ones related to transitive relations (**`:through`** option) the ones related to conditional behavior (**`:conditions`** option), and the ones related to polymorphic behavior (**`:polymorphic`** option). If a given relation r satisfies a given constraint, then we would state that $r \models C$.

3.1.1 Relational Constraints

The Basic Relationships Recall from Chapter 2 that Rails supports three basic types of relations among objects: (1) one-to-one, (2) one-to-many, and (3) many-to-many. We will describe how to formalize these relational constraints using the running example in Figure 2.1.

The `has_one` and `belongs_to` declarations in lines 3 and 10 in Figure 2.1 define a one-to-one relation between the User and Profile classes. More accurately, this is a one-to-zero-or-one relation and it declares that each User object must be associated with zero or one Profile object, and each Profile object must be associated with exactly one User object. In order to formalize this relation as a constraint, let us use o_U and o_P to denote the set of objects for the User and Profile classes and r_{U-P} to denote the relation between User objects and Profile objects. Then the constraint that corresponds to this relation is formalized as:

$$\begin{aligned} &(\forall p \in o_P, \exists u \in o_U, (u,p) \in r_{U-P}) \\ \wedge\ &(\forall p, p' \in o_p, \forall u \in o_U, ((u,p) \in r_{U-P} \wedge (u,p') \in r_{U-P}) \Rightarrow p = p') \\ \wedge\ &(\forall p \in o_p, \forall u, u' \in o_U, ((u,p) \in r_{U-P} \wedge (u',p) \in r_{U-P}) \Rightarrow u = u') \end{aligned} \tag{3.1}$$

Next, let us consider the one-to-many relation between the Profile and Photo classes, which is declared in the Rails data model in Figure 2.1 using the `has_many` and `belongs_to` declarations in lines 11 and 16. Using o_P and o_{Ph} to denote the set of objects for the Profile and Photo classes and r_{P-Ph} to denote the profile-photo relation, the formal data model constraint that corresponds to this declaration is:

$$\begin{aligned}&(\forall ph \in o_{Ph}, \exists p \in o_P,\ (p, ph) \in r_{P-Ph})\\ \wedge\ &(\forall p, p' \in o_P, \forall ph \in o_{Ph}, ((p, ph) \in r_{P-Ph} \wedge (p', ph) \in r_{P-Ph}) \Rightarrow p = p')\end{aligned} \tag{3.2}$$

Finally, Rails allows the expression of many-to-many relations, using the `has_and_belongs_to_many` declaration on both sides of the relation. This is seen in Figure 2.1 on lines 2 and 7 where this declaration is used to set up a many-to-many relation between User and Role. For such declarations we do not have to create any additional constraints since any relation is a many-to-many relation.

Extensions Rails also provides a set of options that can be used to extend the three basic relations. The first option is the `:through` option for the `has_many` and `has_one` declarations. Recall that the `:through` option enables the declaration of new relations that are the composition of two other relations. Consider line 4 in Figure 2.1 which ends with `:through => :profile` and declares a relation between

User and Photo objects. When this declaration is combined with the declarations of the relation between User and Profile objects (lines 3 and 10) and Profile and Photo objects (lines 11 and 16), it specifies that the relation between the User and Photo objects (r_{U-Ph}) is the composition of the relations between the User and Profile objects (r_{U-P}) and the Profile and the Photo objects (r_{P-Ph}). This can be formalized as:

$$\begin{aligned} &\forall u \in o_U, \forall ph \in o_{Ph},\ (u, ph) \in r_{U-Ph} \Leftrightarrow \\ &(\exists p \in o_P,\ (u, p) \in r_{U-P} \wedge (p, ph) \in r_{P-Ph}) \end{aligned} \tag{3.3}$$

The second option that can be used to extend relations is the `:conditions` option, which can be set on all of the four declarations (`has_one`, `has_many`, `belongs_to`, and `has_and_belongs_to_many`). The `:conditions` option limits the relation to those objects that meet a certain criteria. For example, based on the relation declaration in lines 12 and 13 in Figure 2.1, Video objects are only related to a Profile object if their `format` field is `'mp4'`. The formalization of this constraint defines a set of objects ($o_{V'}$) that is a subset of the Video objects (o_V) (corresponding to Video objects with `format` field `'mp4'`) and restricts the relation between the Profile and Video objects (r_{P-V}) to that subset. Formally:

$$o_{V'} \subseteq o_V \wedge (\forall p \in o_P, \forall v \in o_V, (p, v) \in r_{P-V} \Rightarrow v \in o_{V'}) \tag{3.4}$$

Note that since we do not model data fields (such as the `format` field of the Video class), this formalization is necessarily an abstraction.

Rails also supports the declaration of polymorphic associations. In Rails, polymorphic associations are declared by setting the `:polymorphic` option on the `belongs_to` declaration and the `:as` option on the `has_one` or `has_many` declarations. We see the use of the `:polymorphic` option in Figure 2.1 between Tags, Photos and Videos (lines 17, 22, 25). Recall that Photos and Videos do not have a sub-class relationship but they both can have Tags. The use of the `:polymorphic` option in the Tag class creates a relationship which allows any class to act as a Taggable object and relate to the Tag class via this relation. This is formalized by defining a set of objects for the superset (Taggable) and then expressing inheritance using subset constraints. For the example above, we define a new set of objects called Taggable (o_T) and declare that Video objects (o_V) and Photo objects (o_{Ph}) are mutually exclusive subsets of the Taggable objects, as shown below. Then a relation can be formally specified between Tag and Taggable objects using ideas discussed earlier.

$$o_V \subseteq o_T \wedge o_{Ph} \subseteq o_T \wedge o_V \cap o_{Ph} = \emptyset \tag{3.5}$$

3.1.2 Dependency Constraints

The dependency constraints in D are concerned with how a data model instance changes with respect to object deletion. Hence, these constraints express conditions on two consecutive instances of a relation such that deletion of an object from one of them leads to the other instance by deletion of possibly more objects. So in order to determine if a dependency constraint holds, we need two instances of the same relation, say r and r', where one denotes the instance before the deletion, and one denotes the instance after the deletion, respectively. Then, if the pair of relations (r, r') satisfy the dependency constraint, we write $(r, r') \models D$.

The final Rails construct we formalize, the `:dependent` option, creates constraints of this form. In Figure 2.1 we see that the User class has the `:dependent` option set for the relation with the Profile class (line 3). Thus, when a User object is deleted, the Profile object that is associated with that User will also be deleted. Further, since the `:dependent` option is set to `:destroy`, any relations of the Profile class with the `:dependent` option set will cause those associated objects to be deleted as well.

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

$$\begin{aligned}
& o'_P \subseteq o_P \wedge o'_U \subseteq o_U \wedge r'_{U-P} \subseteq r_{U-P} \\
\wedge \quad & (\exists u \in o_U, u \notin o'_U \wedge (\forall u' \in o_U, u' \neq u \Rightarrow u' \in o'_U) \\
\wedge \quad & (\forall p \in o_P, (u,p) \in r_{U-P} \Rightarrow p \notin o'_P) \\
\wedge \quad & (\forall p \in o_P, (u,p) \notin r_{U-P} \Rightarrow p \in o'_P) \\
\wedge \quad & (\forall u' \in o_U, \forall p \in o_P, ((u',p) \in r_{U-P} \wedge (u,p) \notin r_{U-P}) \Rightarrow (u',p) \in r'_{U-P})) \quad (3.6)
\end{aligned}$$

3.2 Formalizing Verification Queries

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

A data model instance is a tuple $I = \langle O, R\rangle$ 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. For each $r_i \in R$ there exists $o_j, o_k \in O$ such that $r_i \subseteq o_j \times o_k$.

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

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

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

Data Model Properties Given a data model $M = \langle S, C, D\rangle$, we will define four types of properties:

1. *State assertions*, denoted by A_S, are properties that we expect to hold for each instance of the data model;

2. *Behavior assertions*, denoted by A_B, 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, are properties we expect to hold in some instance of the data model; and, finally,

4. *Behavior predicates*, denoted by P_B, are properties we expect to hold in some pair of instances that form a behavior of the data model.

We will denote that a data model satisfies an assertion or a predicate as $M \models A$ or $M \models P$, respectively. Then, we have the following formal definitions for these four types of properties:

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

We say that a data model M satisfies a state assertion A_S, denoted by $M \models A_S$ if and only if, for all $I = \langle O, R \rangle$ where I is an instance of M, $R \models A_S$.

We say that a data model M satisfies a state predicate P_S if and only if, there exists an $I = \langle O, R \rangle$ such that I is an instance of M and $R \models P_S$.

We say that a data model M satisfies a behavior assertion A_B if and only if, for all $I = \langle O, R\rangle$, $I' = \langle O', R'\rangle$ where (I, I') is a behavior of M, $(R, R') \models A_B$.

We say that a data model M satisfies a behavior predicate P_B if and only if, there exists $I = \langle O, R\rangle$ and $I' = \langle O', R'\rangle$ such that (I, I') is a behavior of M and $(R, R') \models P_B$.

With these definitions we can now define the *data model verification problem* which is, given one of the above four types of properties, determine if the data model satisfies the property.

As an example of what properties look like, let us say we want to express the following property about the data model in Figure 2.1: Is it possible to have a user with no roles? Formally expressed,

$$\exists u \in o_U, \forall l \in o_L,\ (u, l) \notin r_{U-L} \tag{3.7}$$

where o_U denotes the set of User objects, o_L denotes the set of Role objects, and r_{U-L} denotes the set of related User and Role objects. This property is a state predicate, P_{S1}. To check whether the data model, M_1, satisfies this property, we check $M_1 \models P_{S1}$ according to the definition above.

Our approach to data model verification asks the user to specify such properties that she expects will hold on the data model of her web application. Given a list of such properties, our tool automatically verifies whether these properties hold or not. In the next chapter, we discuss how we perform verification using a bounded approach.

Chapter 4

Bounded Verification

In this chapter we describe our bounded verification approach to automatic data model verification [72].

4.1 Overview

Given the data model verification problem (see Chapter 3), we can solve it using bounded verification, where we check the property for instances within a certain bound. The main idea is to bound the set of data model instances to a finite set, say $\mathcal{I}_k$ where $I = \langle O, R \rangle \in \mathcal{I}_k$ if and only if for all $o \in O$ $|o| \leq k$. Then, given a state assertion A_S (for example), we can check the following condition:

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

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

Similarly, given a predicate P_S, and a bounded set of instances $\mathcal{I}_k$, we can check the condition:

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

If this condition holds we can conclude $M \models P_S$. If the above condition fails on the other hand, we can only conclude that the predicate P_S does not hold for the data model instances in $\mathcal{I}_k$. Bounded verification of behavior assertions and behavior predicates can also be done similarly on bounded data model instances.

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 [54] is a SAT-based bounded verification tool for analyzing object-oriented data models. The Alloy language allows the specification of objects and relations and it allows specification of constraints on relations using first order logic. Alloy Analyzer supports bounded verification of assertions and simulation of predicates which correspond to the assertion and predicate checks we described in the previous chapter.

In order to perform bounded verification of Rails data models, we implemented an automatic translator that translates Active Record specifications to Alloy specifications. After this automated translation, we use the Alloy Analyzer for bounded verification of data model properties. Our approach is summarized in Figure 4.1.

4.2 Translation to Alloy

In this section we describe how we translate Active Record specifications into the Alloy language. The first step of the Active Record to Alloy translation is to map each Active Record class to a `sig` in Alloy, which simply defines a set of objects in the Alloy language.

The Three Basic Relationships When expressing a binary relationship in Alloy, one can give it a multiplicity of `one`, `lone`, `some`, or `set` which correspond to one, zero or one, one or more, and zero or more, respectively. Thus we obtain the mapping of the Rails relationships to Alloy shown in Table 4.1.

Since each relationship declaration in Alloy defines a separate relation, one also has to add a fact block that connects each pair of declarations, constraining them

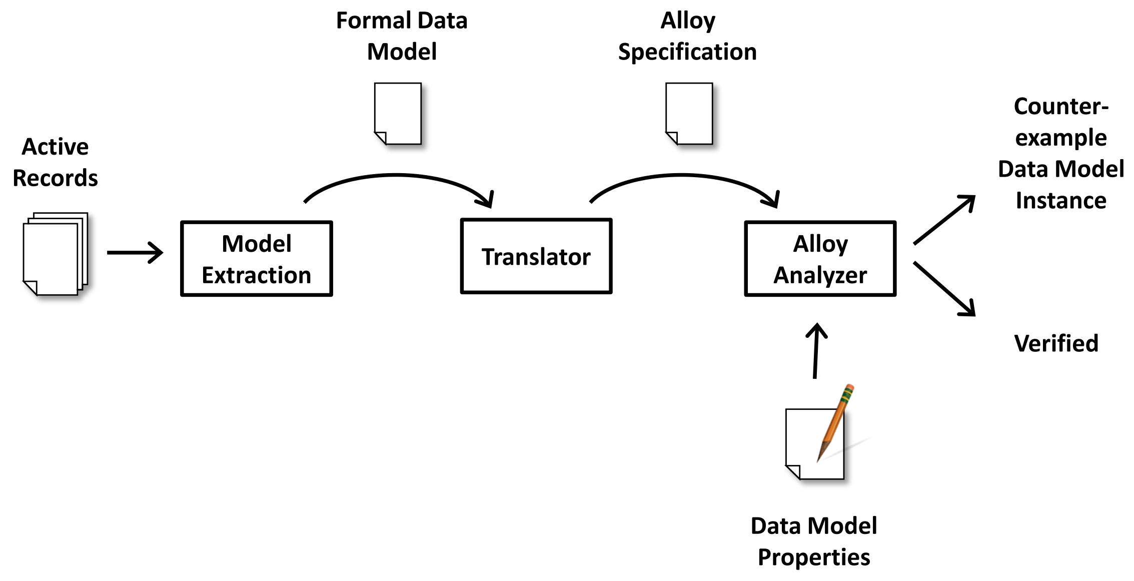


Figure 4.1: Summary of our bounded data model verification approach

Rails Declaration	Alloy Translation

`class ObjectA`
`  has_one :objectB`
`end`
	`sig ObjectA {`
	`  objectB: lone ObjectB`
	`}`

`class ObjectA`
`  has_many :objectBs`
`end`
	`sig ObjectA {`
	`  objectBs:  set ObjectB`
	`}`

`class ObjectA`
`  belongs_to :objectB`
`end`
	`sig ObjectA {`
	`  objectB: one ObjectB`
	`}`

`class ObjectA`
`  has_and_belongs_to_many :objectBs`
`end`
	`sig ObjectA {`
	`  objectBs:  set ObjectB`
	`}`

Table 4.1: Translation of the Rails declaration associations to Alloy

to be inverse relations. For example, the fact block for a one-to-many relationship would look as follows:

```
fact { ObjectA <: objectBs = ~(ObjectB <: objectA) }
```

where `<:` is the domain restriction operation such that `s <: r` contains the tuples in relation `r` that start with an element in `s`, and the operator ∼ is the relational inverse operation where ∼`r` is the inverse of the relation `r`.

The `:through` Option To translate the `:through` option, we follow the mapping in Table 4.1. However, instead of a separate global fact block, we add a local fact block immediately following the signature of the object containing the `:through` declaration. A global fact is not needed for the :through relation because all the fields referred to in the fact refer to the those inside that single signature. So, for the following Rails models:

```
class User < ActiveRecord::Base
  has_one :profile
  has_many :photos, :through => :profile
end
class Profile < ActiveRecord::Base
  has_many :photos
end
class Photo < ActiveRecord::Base
  belongs_to :profile
end
```

the Alloy translation looks as follows:

```
sig User {
   profile: lone Profile,
   photos: set Photo
} { photos = profile.photos }
sig Profile {
   user: one User,
   photos: set Photo
}
```

```
sig Photo {
   profile: one Profile
}
fact {
   User <: profile = ~(Profile <: user)
   Profile <: photos = ~(Photo <: profile)
}
```

The :conditions Option The `:conditions` option means that objects from one class only associate with a subset of objects from another class rather than with the entire set. Thus, to translate the `:conditions` option we create a subset of objects in Alloy which the object with the condition statement can map to. Therefore if we had the following Rails data model:

```
class Profile < ActiveRecord::Base
  has_many :videos, :conditions => "format='mp4'"
end
class Video < ActiveRecord::Base
  belongs_to :profile
end
```

we translate it to Alloy by abstracting the set of addresses for which the condition `format='mp4'` holds, to the set `Mp4_Videos`, as follows:

```
sig Profile { videos: lone Mp4_Video }
sig Video { profile: one Profile
sig Mp4_Video in Video { }
fact {
  Profile <: videos = ~(Mp4_Video <: profile)
}
```

The `in` keyword in Alloy creates a subset; it is used above to create the `Mp4_Video` signature. Since we are not modeling the data fields of the Rails classes, we create this arbitrary subset of `Video` without specifying exactly which elements of `Video` belong in the subset (i.e. the ones which have `'mp4'` as the `format`). The `videos` field in `Profile` can now map to just this subset of `Video`. The global `fact` block establishes this mapping by confirming the `videos` and `profile` fields of the two signatures refer to the same set of objects.

The :polymorphic Option In polymorphic relations, there is a *base* class that can be related to one of many *target* classes. Moreover, this relationship is expressed via a single field in the base class. So, to translate the polymorphic relation, we need to enclose the target classes inside a single supertype which the relation in the base class can refer to. However the translation for polymorphic relations is not straightforward since a target class can have polymorphic relations with multiple classes. Modeling these kinds of scenarios requires multiple inheritance.

To understand how to simulate multiple inheritance in Alloy, let us assume that the class `Photo` needs to inherit from both `Taggable` and `Addressable`. In order to simulate multiple inheritance, all Active Record classes are made a subset of some other superclass, say `ActiveRecord`. We will use the `extends` keyword in Alloy to ensure the subsets are disjoint. Then statements are added to the global `fact` block which will say `Photo` is a subset of both `Taggable` and `Addressable`; but this time we will use the `in` keyword to declare the subset, which will allow overlapping (as opposed to the `extends` keyword, which forces the subsets to be disjoint).

Let us take a look at a concrete instance from the running example in Figure 2.1. Below are a set of Rails models with a polymorphic association. `Tag` has a `taggable` association that both `Photo` and `Video` refer to:

```
class Tag < ActiveRecord::Base
  belongs_to :taggable, :polymorphic => true
end
class Photo < ActiveRecord::Base
  has_many :tags, :as => :taggable
end
class Video < ActiveRecord::Base
  has_many :tags, :as => :taggable
end
```

The first step in translating these models is to create a common base class that all classes extend. This common base class will allow the multiple inheritance to be simulated. The natural choice is to call it `ActiveRecord`, so we will add the following Alloy signature to all Alloy specifications:

```
  abstract sig ActiveRecord { }
```

The `abstract` keyword tells Alloy that this signature has no elements except those belonging to its extensions. All signatures will either inherit from this class or the parent class if one is specified in the corresponding Rails model.

The next step is to create a supertype for the target classes to be enclosed in. The supertype will be called `Taggable` and it will contain the `has_many` relation, translated as described earlier:

```
sig Taggable in ActiveRecord {
   tag: set Tag
}
```

Next, the relationship between `Taggable` and the target classes (`Photo` and `Video`) will be established via `fact`s. Specifically, we will state that the target classes are subsets of `Taggable`, using the `in` keyword.

Further, Alloy does not allow subsets to be `abstract` if the superset is `abstract`, like we have made `ActiveRecord`. Thus we will also have to specify as `fact`s that there are no elements in `Addressable` except those belonging to the target classes. Finally, since our design requires *all* signatures to extend `ActiveRecord`, we also have to add `fact`s to state that `Taggable` is disjoint from all other non-target classes in `ActiveRecord`. The final Alloy translation is given below.

```
abstract sig ActiveRecord {}
sig Taggable in ActiveRecord {
   tag: set Tag
}
```

```
sig Tag extends ActiveRecord {
   taggable:  one Taggable
}
sig Photo extends ActiveRecord {}
sig Video extends ActiveRecord {}
fact {
   Photo in Taggable
   Video in Taggable
   all x:  Taggable | x in Photo or x in Video
   no Tag & Taggable
}
```

4.3 Translating the Dependency Constraints to Alloy

In this section, we translate the `:dependent` option, which specifies what behavior to take on deletion of an object with regards to its associated objects. To incorporate this dynamism, the model must allow analysis of how sets of objects and their relations *change* from one state to the next. Thus we need a slightly different translation algorithm from the one we have been presenting so far.

In order to handle the `:dependent` option, we will be creating invokable constraints, or `predicates` in Alloy, which will model the deletion of an object. We will also need Alloy signatures to represent the state of a data model instance, i.e. the set of all objects and their relations. In particular, we will have a `PreState` signature to represent the state of objects before the deletion operation, and a `PostState` signature to represent the state after the deletion. We can then use these signatures to check whether some invariant holds after an object is deleted.

Basic Translation We will use snippets of the running example to explain each piece of this new translation algorithm. Let us begin with the following portion of the Rails data model from the running example:

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

As before, the Alloy specification for this model will contain a signature for each class. It will also contain a `PreState` and a `PostState` signature, as just discussed. Since the `PreState` and `PostState` signatures represent the whole data model instance, they will need references to all object types and relations. Thus we obtain the following Alloy specification:

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

The `PreState` sig contains fields `profiles` and `users` to hold objects of each type in the system. Next, it contains a field `relation` to hold the related `Profile` and `User` objects. The product operator, `->`, produces a mapping between `Profile`s and `User`s. The multiplicity keyword `lone` tells Alloy that `relation` maps each `User` object to zero or one `Profile` object, and the keyword `one` tells Alloy that every `Profile` object is mapped to exactly one `User` object. Note that in the translation of relations, the multiplicity keywords are the same as the ones used in the earlier translation summarized in Table 4.1 (e.g. `belongs_to :user` produces `one User` and `has_one :profiles` produces `Profile lone`). Also note the `one` preceding `sig PreState`. This tells Alloy that there will be exactly one instance of `PreState` in any data model instance.

The definition of `PostState` is exactly the same. The only difference is that its relations always map a `set` of objects to another `set` of objects. The reason to not specify the relation cardinalities here as well is because when the cardinality is `one`, it forces the mapping to be total. However once an object has been deleted, we need to remove it from the relation, causing the need for a partial mapping in the `PostState`. Since the relations in `PostState` will be defined in the delete predicates using the `PreState` relations, the cardinalities among the remaining (live) objects will be preserved.

Let us now turn to the definition of the delete predicates. As an example, let us generate the predicate that deletes a `Profile` object. To start, we define the `deleteProfile` predicate to accept a `PreState` object, a `PostState` object and a

`Profile` object as parameters. The body of the predicate begins by stating that `s`, the `PreState` object, contains all existing objects:

```
pred deleteProfile [s: PreState, s': PostState, x: Profile] {
   all x0: Profile | x0 in s.profiles
   all x1: User | x1 in s.users
```

Finally, we describe the data model instance after the deletion:

```
   s'.profiles' = s.profiles - x
   s'.users' = s.users
   s'.relation' = s.relation - (x <: s.relation)
}
```

Here we have deleted `x`, a `Profile` object, by removing it from the set of `Profile` objects in `PostState`. We have also updated the relation by setting the `PostState` relation to be the `PreState` relation minus all the tuples whose domain is `x` (using the scoping operator <: described earlier). This removes all of `x`'s relations from `relation'`.

It is important to note here that the relation is only updated if it is a `:belongs_to` or `:has_and_belongs_to_many` relationship in the Rails model. (So in the delete predicate for `User` which contains the `has_one` declaration, `relation` would remain unchanged: `s'.relation' = s.relation`.) This is due to the way the relationships are implemented in Rails. In the database, the foreign key is stored with the object that has the `:belongs_to` relationship (for the one-to-one and one-to-many relations) or in a join table for the `:has_and_belongs_to_many` relationships. Thus, an object's `has_one` and `has_many` relations are not affected when an object is deleted. Note that deleting an object on the `has_one` or `has_many` side may cause a dangling reference if the `:dependent` option is not set; our model can be used to check for such cases (see examples in our experiments chapter, Chapter 6).

The `:through` Option Next, let's analyze the following partial Rails model to understand how to translate the `:through` option for the dynamic Alloy specification.

```
class User < ActiveRecord::Base
  has_one :profile, :dependent => :destroy
  has_many :photos, :through => :profile
end
class Profile < ActiveRecord::Base
  belongs_to :user
  has_many :photos
end
class Photo < ActiveRecord::Base
  belongs_to :profile
end
```

The basic setup for the Alloy specification is the same: a signature for each class, a `PreState` and `PostState` signature, each with a field for every set of objects and relations between them.

```
sig User {}
sig Profile {}
sig Photo {}
one sig PreState {
   users:  set Users,
   profiles: set Profile,
   photos: set Photo,
   relation1: User one -> lone Profile,
   relation2: Profile one -> set Photo,
   thru_relation = relation1.relation2
}
one sig PostState {
   users': set Users,
   profiles': set Profile,
   photos': set Photo,
   relation1': User set -> set Profile,
   relation2': Profile set -> set Photo,
   thru_relation' = relation1'.relation2'
}
```

The new idea is the translation of the relation with the `:through` option set, i.e. the one between `User` and `Photo`. We use the join operator, `.`, to define `thru_relation` to be the join of the other two relations. Delete predicates would also be produces for every object type in the data model, defining each set of objects and relations after the deletion according to all uses of the `:dependent` option. If we wanted to define the delete predicate for `User` in the example above, it would look as follows:

```
pred deleteUser [s:  PreState, s':  PostState, x:  User] {
    all x0:User | x0 in s.users
    all x1:Profile | x1 in s.profiles
    all x2:Photo | x2 in s.photos
    s'.users' = s.users - x
    s'.profiles' = s.profiles - x.(s.relation1)
    s'.photos' = s.photos
    s'.relation1' = s.relation1 - (s.relation1 :> x.(s.relation1))
    s'.relation2' = s.relation2
}
```

In this example, when a user is deleted, it also deletes the associated Profile object since the `:dependent` option is set. Since no relations in the Profile class have the `:dependent` option set, the delete is not propagated further. Relations are only deleted if the object on the `:belongs_to` side of the relation is deleted. In this case, the Profile class contains a `:belongs_to` relation. Thus the relation between Profile and User is updated such that the user that was deleted and its associated profile are removed. The delete predicate above says exactly this. It starts off by stating all objects in the universe exist in the PreState. The second part defines the set of objects in the PostState: the deleted user, `x`, is removed from `users'`, the profile object associated with that user is also deleted, but the set of photos stay the same as in the PreState. In the third and final part of the delete predicate, the relations in the PostState are defined. The relation between User and Profile, `relation1'`, is updated to remove the tuple containing the deleted user, `x`. The other relation, `relation2'`, between Profile and Photo remains the same as it was in the PreState. Note that `thru_relation'` does not need to be updated explicitly; it will be updated automatically because it is defined using `relation1'` and `relation2'`.

The `:conditions` and `:polymorphic` Options The translation of the `:conditions` and the `:polymorphic` options remain the same as described in the previous translation, except that they contain the `PreState` and `PostState` sigs. Further, the definitions of the delete predicates will take into account usage of the `:dependent` option.

4.4 Automated Translator Implementation

We implemented a translator that translates data models in Rails applications to Alloy. The first step of the translation is parsing the Rails model files (i.e., Active Record files). We do this using a parser written in and for Ruby, called

ParseTree [80]. ParseTree extracts the parse tree for an entire Ruby class and returns it as an s-expression [82]. S-expressions are generated for each model file that contains a class that inherits from `ActiveRecord`. We then create an s-expression processor to traverse the generated s-expressions and translate them to a single Alloy specification file. The translation occurs according to the principles discussed in this chapter. This automated translator is joined with the Alloy Analyzer to perform one of the two main functions of our data model verification tool, namely bounded verification of data models. The other main function is performing unbounded verification, which we present next.

Chapter 5

Unbounded Verification

Another way to solve the data model verification problem is to use unbounded verification. In particular, we use the formalization presented in Chapter 3 to convert verification queries about the data model to satisfiability of formulas in the theory of uninterpreted functions. We then use an SMT solver to answer the verification queries. This approach [73] is summarized in Figure 5.1. In this chapter, we describe the translation of Active Records to the theory of uninterpreted functions using the syntax of SMT-LIB. We also present our data model projection algorithm which reduces the size of SMT-LIB specifications based on the property provided.

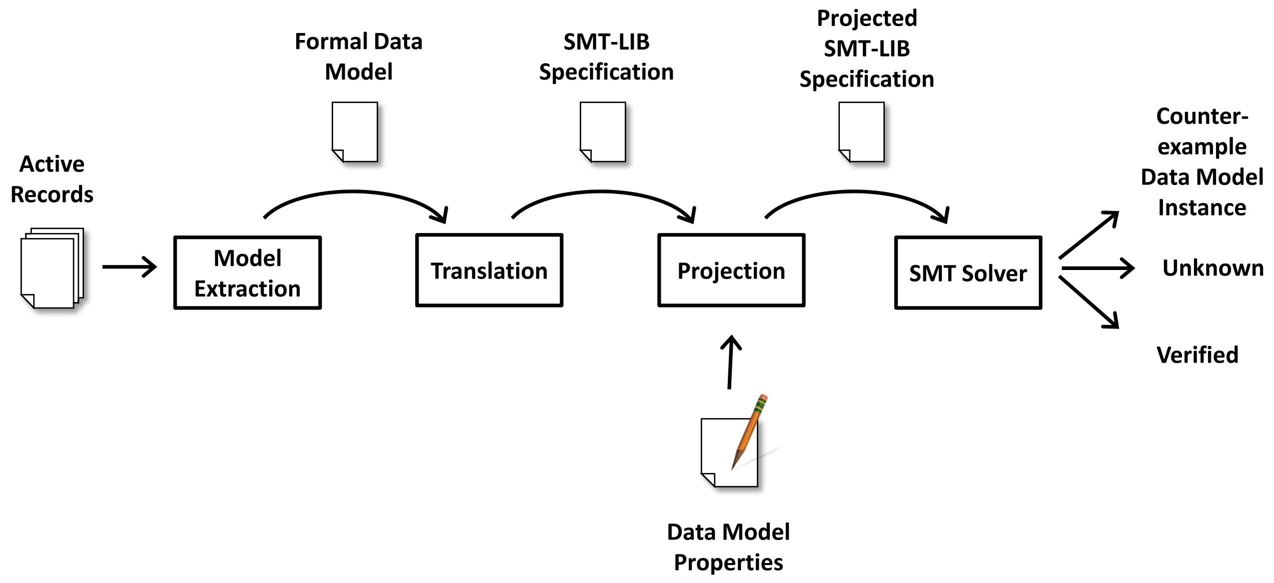


Figure 5.1: Summary of our unbounded data model verification approach

5.1 Translation to SMT-LIB

SMT-LIB [88] is the standard input language for SMT solvers. We have implemented a translator that takes Rails Active Record files as input and generates an SMT-LIB specification for the data model. The generated SMT-LIB specification consists of a conjunction of constraints in the theory of uninterpreted functions with quantification. In this section we describe how the various Active Record constructs which define the data model can be translated to constraints in the theory of uninterpreted functions.

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

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

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

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

After this short overview of the SMT-LIB language syntax, we now explain how we translate the formal model constraints discussed in Chapter 3 to SMT-

LIB. Let us first consider constraint (3.2) from Chapter 3 which characterizes the semantics of a one-to-many relation declaration. We translate the one-to-many relation between the Profile and Photo classes to SMT-LIB using an uninterpreted function as:

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

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

Constraint (3.1) in Chapter 3 represents the semantics of a one-to-one relation declaration. We translate such a relation to SMT-LIB using an uninterpreted function like above, but adding an extra constraint restricting the cardinality of the relation. For example, the one-to-one relation between User and Profile is translated as:

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

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

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

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

As discussed in Chapter 3, relations that are the composition of other relations can be declared in a data model using the `:through` keyword and constraint (3.3) provides a formalization of such declarations. For example, assume that Users are associated with Profiles, Profiles are associated with Photos and the data model declares a third relation between Users and Photos such that it is the composition of the other two relations. This is translated to SMT-LIB as[1]:

```
(declare-sort Profile 0)

(declare-sort Photo 0)

(declare-sort User 0)

(declare-fun profile_photo (Photo) Profile)

(declare-fun user_profile (Profile) User)

(declare-fun user_photo (Photo) User)

(assert (forall ((u User)(ph Photo))

    (iff  (= u (user_photo ph))  (exists ((p Profile))

    (and (= u (user_profile p)) (= p (profile_photo ph))  ))

) ))
```

Next, the `:conditions` option is used to express a relationship between one set of objects and the subset of another set of objects and is formalized in constraint (3.4) of Chapter 3. Since there is no support for subtyping or inheritance in the SMT-LIB language, we model the `:conditions` option by creating a boolean function that returns true if and only if the argument object is in the designated subset. To give a concrete example, say Photos are associated with a Profile only if the Profile is active. The SMT-LIB translation of such a declaration would be:

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

```
(declare-sort Photo 0)
(declare-sort Profile 0)
(declare-fun isActive (Profile) Bool)
(declare-fun activeprofile_photo (Profile) Photo)
(assert (forall ((p Profile)( ph Photo))
  (=> (= ph (activeprofile_photo p)) (isActive p) )
))
```

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

Next, Rails Active Records support the specification of polymorphic relations as formalized in constraint (3.5). In order for one type to be related to multiple other types, one can create a supertype that the former type can relate to. For example, if a Tag can be related to both Photos and Video, we can create a supertype of Photo and Video that Tag can be related to. Let us call this supertype Taggable and let Photo and Video be subtypes of it. As mentioned earlier, SMT-LIB does not support subtyping so we use boolean functions to model such a declaration. We also add a constraint that states the Taggable type is abstract, i.e. all Taggable objects are either Photos or Videos, and that these subtypes are mutually exclusive:

```
(declare-sort Tag 0)

(declare-sort Taggable 0)

(declare-fun isPhoto (Taggable) Bool)

(declare-fun isVideo (Taggable) Bool)

(assert (forall ((t Taggable)) (and

    (or (isPhoto t) (isVideo t) )

    (iff (isPhoto t) (not (isVideo t)) )

)))

(declare-fun taggable_tag (Tag) Taggable)
```

This example shows a simple case of polymorphic relations. In general, a class may be polymorphically-related to multiple classes. For instance, Multimedia may have a polymorphic relation with the Video and Audio classes. Combined with the scenario above, Video will now require two supertypes (say Taggable and MultimediaItem). In our tool we actually create one ultimate supertype called PolymorphicClass of which any polymorphically-related types are subtypes (such as Photo, Video, and Audio) as well as their supertypes (Taggable and MultimediaItem). All these subtypes are expressed in SMT-LIB language using boolean functions. Then an `assert` is added that constrains which types are subtypes of which supertypes, that subtypes are mutually exclusive of others in the same supertype, that the supertypes themselves are abstract (meaning all elements belong to one of its subtypes), and that PolymorphicClass is also abstract. Furthermore, since subtypes (such as Photo and Video) are not types of their own (i.e. no sort is declared for them since they are of type PolymorphicClass), any non-polymorphic relations with these classes require an assert that enforces the range of the function, similar to what we did for constraint (3.4).

Finally, we discuss delete dependencies that are declared using the `:dependent` option and formalized like constraint (3.6) in Chapter 3. This type of constraint expresses a change from one state of the model (before an object is deleted) to another (the state of the model after the object deleted, i.e., post-delete state). We model the post-delete state in SMT-LIB translation using boolean functions (denoted with the prefix "Post"). There is one such function for every type. This function returns true if the object exists after the delete operation. For example:

```
(declare-sort User 0)

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

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

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

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

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

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

5.2 Data Model Projection

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

We formally define the property-based data model projection as a function, denoted by Π, that takes a data model and a property as input and returns a new data model. Hence, given a data model $M = \langle S, C, D \rangle$ and a property p, $\Pi(M, p) = M_p$ where $M_p = \langle S, C_p, D_p \rangle$ is the projected data model such that $C_p \subseteq C$ and $D_p \subseteq D$. Note that the projection function removes some of the relational and dependency constraints from the model, therefore reducing the size of the projected model. If the property p is a state assertion or a state predicate (denoted by A_S and P_S in Chapter 3), then the projection function Π removes all the dependency constraints (i.e., $D_p = \emptyset$) since dependency constraints are only relevant for behavior assertions and predicates (denoted by A_B and P_B in Chapter 3).

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

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

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

In this chapter, we discussed our unbounded verification approach and data model projection algorithm. In the next chapter we describe the experiments performed using both the unbounded approach presented in this chapter, and the bounded verification approach presented in the previous chapter.

Algorithm 1 Data Model Projection Algorithm (part 1 of 2)

Input: *model*: Rails Active Records files; *p*: property; *delclass*: the class name for the deleted object (only needed when *p* is a behavior assertion or predicate)

Output: Projected SMT-LIB specification

```
pclasses := list of classes mentioned in p
prelations := list of relations mentioned in p

if p is a behavior assertion or predicate then
    Follow dependencies for delclass with respect to the relations given in prelations
    Add any dependent classes, and the relations through which they are dependent, to pclasses
and prelations
end if

for all class in pclasses do
    Output an uninterpreted function declaration and Post function declaration for class
    if there exists a relation in prelations that has a conditional relation with class then
        Output a boolean function declaration that models the conditional subset
    end if
end for

Output the polymorphic constraints for the polymorphic classes in pclasses
```

Algorithm 1 Data Model Projection Algorithm (part 2 of 2)

```
for all relation in prelations do
    Output a function declaration, any associated constraints based on the declaration of the relation, and the Post function declaration for the relation
    if relation is a transitive relation that is the composition of multiple relations then
        Output function declarations, associated constraints, and Post function declarations for all relations that are part of the composition
    end if
end for
```

Chapter 6

Experiments

In this chapter we discuss the results of our experiments for evaluating the effectiveness of our SAT-based bounded and SMT-based unbounded data model verification approaches.

6.1 Overview

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

We used Microsoft's SMT solver, Z3 [100], in our experiments. In addition to returning unsatisfiable or satisfiable, an SMT solver may also return “unknown” or it may timeout since the quantified theory of uninterpreted functions is known to be undecidable [11]. In our experiments the timeout limit was set to five minutes.

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

6.2 The Applications

Table 6.1 lists the sizes of the five applications in terms of lines of code, the number of classes, and the number of data model classes. LovdByLess [63] is a social networking site with the usual features, such as user profiles with pictures, leaving messages, and becoming friends. Tracks [93] is an application that lets users create and manage to-do lists, where lists can be organized by context and project. OpenSourceRails (OSR) [77] is a social project gallery application that allows users to submit projects, as well as bookmark and rate them. FatFreeCRM [35] is a customer relationship management software that organizes a business' customers, campaigns, opportunities, and accounts. Substruct [91] is an e-commerce application where users can add products to cart and create wishlists.

Table 6.1: Sizes of the Applications

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

6.3 Verification Results

The properties we checked on these applications are listed in Table 6.2, along with their type from Section 3.2. (A_S for state assertions, A_B for behavior assertions, P_S for state predicates, and P_B for behavior predicates).

The results of the verification are also shown in Table 6.2. ✓ indicates that the property passed verification and $\times$ indicates that the property failed. A total of sixteen properties we tried to verify failed. We investigated each of these failures manually to determine if they corresponded to data model errors.

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

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

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

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

Table 6.2: Verification Results

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

6.4 Performance

To further evaluate the effectiveness of our approach, we measured performance of the verification task. Specifically, we measured the verification time reported by Z3 and Alloy, as well as the number of variables and clauses produced in the SMT and Alloy specifications.

Measurements were taken for the specifications generated for each property. The values were then averaged over the properties for each application. The results for the verification times for the SMT-LIB specifications using Z3 are given in Table 6.3. What we immediately noticed is that the verification is extremely fast; the longest verification time is just 0.025 seconds. One thing to note is that these values do not include the times for those properties that timed out during verification. There were four such properties, all for FatFreeCRM. So although unbounded verification is very quick, the disadvantage is that some properties may not give an answer to the verification query.

The difficulty the SMT solver is having when it times out is due to the number of quantifiers in the SMT specification. To minimize this number we ran the experiments again, this time using the data model projection algorithm discussed in Section 5.2. With projection we were able to obtain answers to all of the properties that timed out. Furthermore, the verification time decreased for all properties, as shown in Table 6.3. On average the verification time decreased by 40% after using data model projection.

Table 6.3: Z3 Verification Times in seconds

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

To compare effectiveness between bounded and unbounded verification, the same measurements were taken as with Z3, but for Alloy over an increasing bound from at most 10 objects for each class to at most 35 objects for each class. These values were plotted alongside the Z3 verification time for each application. As seen in Figure 6.1, unbounded verification is much faster than bounded verification, even for the smallest bound of 10 objects. Bounded verification using Alloy took up to tens of seconds whereas Z3, took less than a second.

Since Alloy specializes in the verification of object models, it is rather surprising that there is such a drastic difference between the verification times of

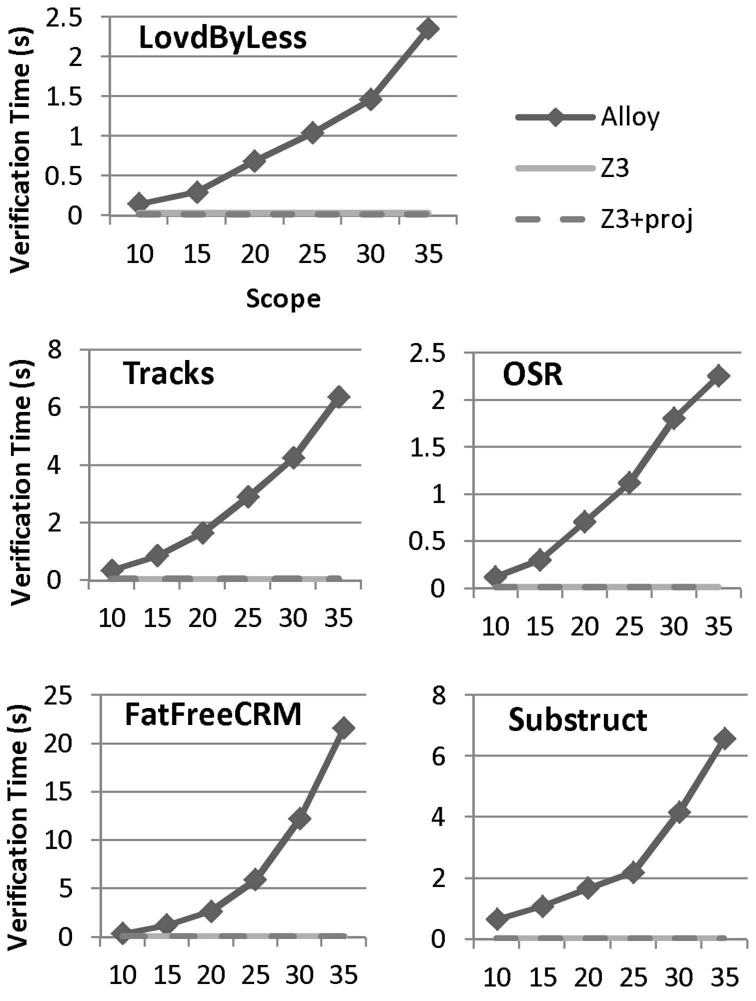


Figure 6.1: Verification Time, Alloy vs Z3

Z3 and Alloy. There may be several reasons why unbounded verification did so well. Z3 uses many heuristics to eliminate quantifiers in formulas. It uses an E-graph to instantiate quantified variables which, in conjunction with code trees, an inverted path index and eager instantiation, makes it very effective at dealing with quantifiers [21]. (Note that these heuristics do not affect the soundness of the verification.) Another reason why Z3 performed better than Alloy may be due to their implementation languages: Z3 is implemented in C++ whereas Alloy (as well as the SAT solver it uses, SAT4J) is implemented in Java. Finally, another likely reason that Z3 is more efficient than Alloy is that SMT solvers operate at a higher level of abstraction than SAT solvers. Thus SMT solvers can use information about the structure and semantics of a formula to make inferences about satisfiability more accurately as well as more efficiently than a SAT-based approach which converts the verification to SAT formulas using a Boolean encoding. In fact, due to the increasing size of the Boolean encoding, bounded verification suffers from an exponential increase in verification time with increasing bound.

Besides verification time, we also measured the number of clauses and variables created by Alloy's SAT translation. These measurements were averaged over the properties for each application and plotted over increasing scope, as shown in Figure 6.2.

The number of variables and clauses were also averaged over properties for the SMT-LIB specifications for each application. By number of variables we mean the number of sorts, functions, and quantified variables in the SMT-LIB specification. By number of clauses we mean the number of asserts, quantifiers and operations. Figure 6.3 displays is a plot of the number of clauses and variables in the SMT-LIB specifications for each application. Both the projected and non-projected versions of the specifications are shown. We see a tremendous 80% decrease in the number of variables and clauses after performing the data model projection.

Although the number of variables and clauses in the SMT specification are not directly comparable to the figures produced by Alloy, we can still observe that the SMT formula size is much smaller than the one used by the SAT solver. We also observe that bounded verification has the disadvantage that the size of the formula used by the SAT solver increases exponentially with respect to bound.

To summarize, our experimental results indicate that unbounded verification using SMT solvers is more efficient than bounded verification. However, since the unbounded approach is not guaranteed to terminate because we are generating SMT-LIB specifications in the undecidable theory of uninterpreted functions with quantification, we observe that bounded and unbounded verification can be complementary approaches since bounded verification can be used when the unbounded approach fails. Overall, our experiments demonstrate the approaches we presented in the previous two chapters is a feasible and efficient approach to data model verification.

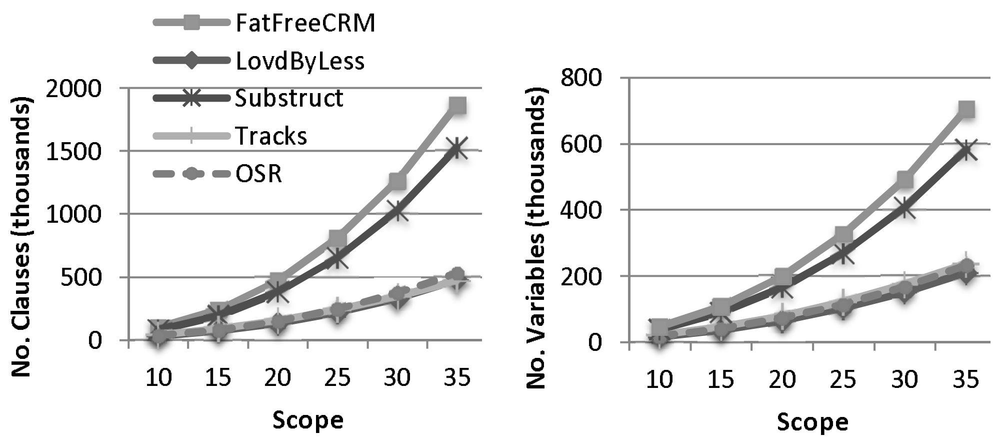


Figure 6.2: Formula Size, Alloy

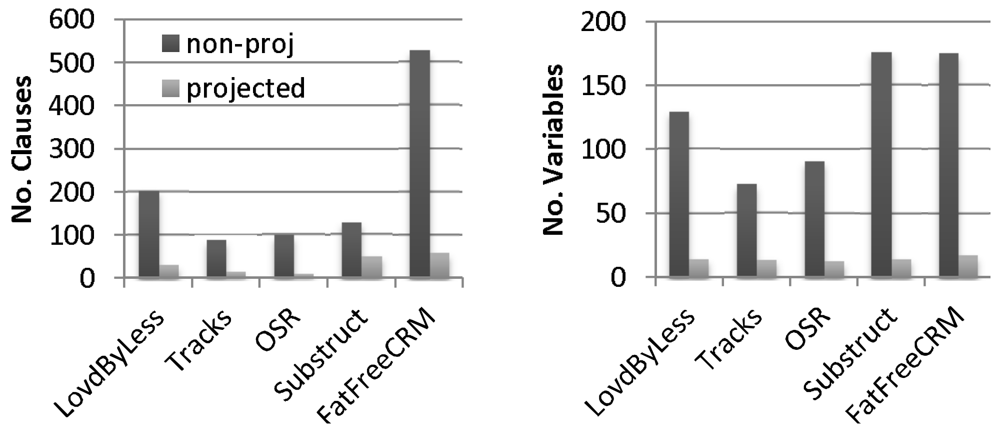


Figure 6.3: Formula Size, SMT

Chapter 7

Automated Property Inference and Repair

In the verification methods presented so far, the user is required to manually specify the properties she wishes to check about her application's data model. In order to further automate the process of verifying data models, we can automate the process of property-writing. This chapter describes our work on automatically discovering certain classes of data model properties [74].

7.1 Overview

In the techniques presented earlier in Chapters 4 and 5, the effectiveness of finding data model errors is highly dependent on the quality of the properties specified by the user. Since the manual specification process can be time-consuming, error-prone and lack thoroughness, errors can be missed during verification. Another difficulty of writing properties as that the user may not be familiar with the modeling language in which the properties need to be written (i.e. Alloy and SMT-LIB).

To address these obstacles, we propose techniques that automatically infer properties about the data model of a web application. The techniques infer properties based on the structure of the data model schema extracted from the object-relational mapping of an application. The schema can be viewed as a graph whose nodes are the object classes and whose edges are relations. We developed heuristics that explore the structure of this graph and look for a set of patterns.

Based on such patterns we infer a set of properties about the data model that we expect to hold, thus assisting the developer in the specification of data model properties. Once the automatically inferred properties are generated, a data model verification technique (such as the ones presented in Chapters 4 and 5) can be used to determine if the inferred properties are actually enforced by the data model.

Finally, our approach includes techniques to automatically generate repairs for the properties that fail. These repairs are suggested modifications to the data model that establish the inferred properties.

7.2 The Formal Data Model Schema

We begin by formalizing a data model as a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{C}, \mathcal{D} \rangle$ where $\mathcal{S}$ is the data model schema identifying the sets and relations of the data model, $\mathcal{C}$ is a set of relational constraints, and $\mathcal{D}$ is a set of dependency constraints.

The schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ identifies the object classes $\mathcal{O}$ and the relations $\mathcal{R}$ in the data model. In the schema, each relation is specified as a tuple containing its domain class, its name, its type and its range class where $\mathcal{R} \subseteq \mathcal{O} \times N \times T \times \mathcal{O}$, N is a string denoting the name of the relation, and
$T = \{\text{zero-one, one, many}\} \times \{\text{zero-one, one, many}\} \times$
$\{\text{conditional, not-conditional}\} \times \{\text{transitive, not-transitive}\} \times$
$\{\text{polymorphic, not-polymorphic}\}$
is the set of relation types, which are a combination of type qualifiers denoting the cardinality of the domain and range of the relation, whether the relation is conditional or not, whether it is transitive or not and whether it is polymorphic or not. Not all combinations of these attributes are allowed in relation declarations.

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

As an example, take a look at the data model shown in Figure 7.1. This simplified Active Records specification is based on an open source Ruby on Rails application called Tracks. This application allows users to create todo lists. Todos are organized by contexts (such as school, work, home), and todos can be tagged. A set of preferences is saved for each user.

The schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ for the example data model will consist of the following set of object classes $\mathcal{O} = \{$User, Preference, Context, Todo, Tag$\}$ and $\mathcal{R}$ will contain five tuples, one for each relation defined in Figure 7.1: User-Preference, User-Context, User-Todo, Context-Todo, Todo-Tag. For example, the tuple for the User-Preference relation will be: (User, User-Preference, (one, one-zero, conditional, not-transitive, not-polymorphic), Preference).

In Figure 7.2 we give a graphical representation of the schema extracted from the data model in Figure 7.1. The rectangular nodes in the graph correspond to the object classes and the edges represent the relations. The graphical representation of the edges differ based on the type of the relation that they represent, as explained in Figure 7.3.

```
class User < ActiveRecord::Base
     has_one :preference,
             :conditions => "is_active=true"
     has_many :contexts
     has_many :todos
end
class Preference < ActiveRecord::Base
     belongs_to :user
end
class Context < ActiveRecord::Base
     belongs_to :user
     has_many :todos, :dependent => :delete
end
class Todo < ActiveRecord::Base
     belongs_to :context
     belongs_to :user
     has_and_belongs_to_many :tags
end
class Tag < ActiveRecord::Base
     has_and_belongs_to_many :todos
end
```

Figure 7.1: A simplified data model based on a web application called TRACKS that manages todo lists.

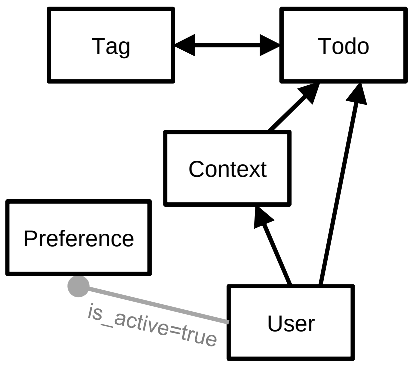


Figure 7.2: The data model schema extracted from the data model shown in Figure 7.1.

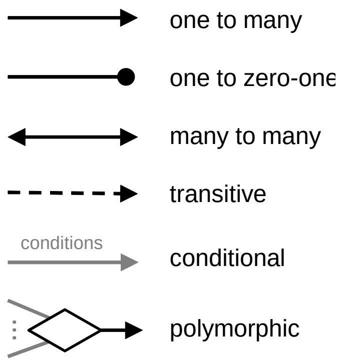


Figure 7.3: Graphical representations of the relation types.

The relational constraints, $\mathcal{C}$, in a formal data model express the constraints on relations that are imposed by their declarations. For example, lines 4 and 11 in Figure 7.1 declare a one to many relation between the User and Context objects. In order to formalize this cardinality constraint, let us use o_U and o_C to denote the set of objects for the User and Context classes and r_{U-C} to denote the relation between User objects and Context objects. Then the constraint that corresponds to this relation is formalized as:

$$
\begin{aligned}
&(\forall c \in o_C, \exists u \in o_U, (u, c) \in r_{U-C}) \\
\wedge\ &(\forall u, u' \in o_U, \forall c \in o_C, ((u, c) \in r_{U-C} \wedge (u', c) \in r_{U-C}) \Rightarrow u = u')
\end{aligned}
\tag{7.1}
$$

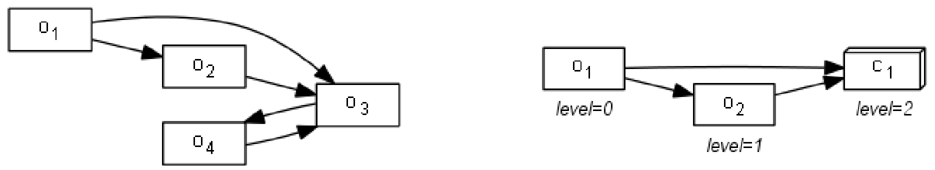
Semantics of all of the data model declaration constructs we discussed above other than the dependency constraints can be formalized similarly (see Chapter 3). The dependency constraints are more complex due to the fact that they require additional modeling of the delete operation.

7.3 Property Inference

Below we present three heuristics for inferring three types of properties using the data model schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ defined in the previous section. These are the types of properties we encountered the most during our manual analysis of data models in the work described in the previous chapters. The heuristic algorithms take as input the data model schema $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$ and output a list of properties.

7.3.1 Delete Propagation

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



(a) (b)

Figure 7.4: A sub-schema and the corresponding acyclic graph constructed during the Inference Algorithm for Delete Propagation.

object class o is deleted then all objects from the object class o' that are related to the deleted object via the relation r are also deleted.

The heuristic for this property type first obtains a sub-schema by removing all relations in $\mathcal{R}$ that are transitive or are 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 7.4a. Cycles in this graph are removed by collapsing strongly connected components to a single node. For the schema in Figure 7.4a, nodes o_3 and o_4 are collapsed to a single node called c_1 in Figure 7.4b. 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 7.4a are given in Figure 7.4b. As can be seen, node o_1 is assigned level 0 since it has no incoming edges. The remaining nodes are assigned levels as just described.

The *deletePropagates* property is inferred if the difference in levels between the nodes a relation connects is not greater than one. The intuition here is that if the difference between the levels of the nodes is greater than one, then there could 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 2.

7.3.2 Orphan Prevention

The next heuristic infers properties about preventing orphaned objects. An orphaned object results after a delete operation if there is an object class related to a single other object class. An object becomes orphaned when the object it is related to is deleted but the object itself is not. Orphan chains can also occur, which begin with an object class that is related to a single object class, and continue with object classes that are related to exactly two object classes, one of which is the previous object class in the chain. Consider an object of the final class of a chain, such as o_{m-1} in Figure 7.5. When the object it is related to (of the class o_m) is deleted but the object itself is not (of the class o_{m-1}), the entire chain

Algorithm 2 Inference Algorithm for Delete Propagation

Input: Data model schema, $\mathcal{S} = \langle \mathcal{O}, \mathcal{R} \rangle$

Output: List of inferred properties

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

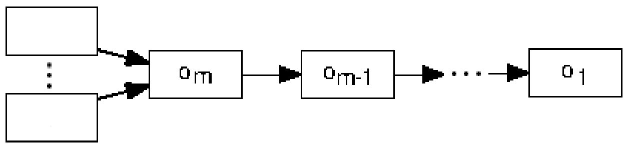


Figure 7.5: The pattern used for recognizing orphan chains.

of objects (of classes $o_{m-1}, ..., o_1$) becomes orphaned. We state the property which asserts that there are no orphans as *noOrphans*(r), where $r = (o, t, n, o') \in \mathcal{R}$ is a relation. Specifically, this property asserts that deleting an object from object class o' does not leave any orphaned objects in class o, or orphan chains that begin with an object in class o.

The heuristic that infers this property looks for potential orphans or orphan chains by analyzing the directed graph that corresponds to the sub-schema which is obtained from the original data model schema by removing all relations in $\mathcal{R}$ that are not one to many or one to zero-one. The orphan prevention property inference algorithm is shown in Algorithm 3.

Algorithm 3 Inference Algorithm for Orphan Prevention

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

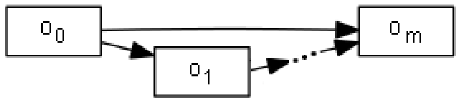


Figure 7.6: The pattern used for inferring transitive relations.

7.3.3 Transitive Relations

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

The heuristic for this property defines a sub-schema by removing all relations in $\mathcal{R}$ that are polymorphic, transitive, conditional or many to many. The algorithm looks for paths of relations of length more than one. If there also exists an edge connecting the first node in the path to the last node, then the algorithm infers that this edge should be a transitive relation. The intuition here is that if there are multiple ways to navigate relations between two classes, the composition of the relations corresponding to alternative ways of navigation should be equivalent. The pattern used for this heuristic is shown in Figure 7.6. 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 4.

Algorithm 4 Inference Algorithm for Transitive Relations

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

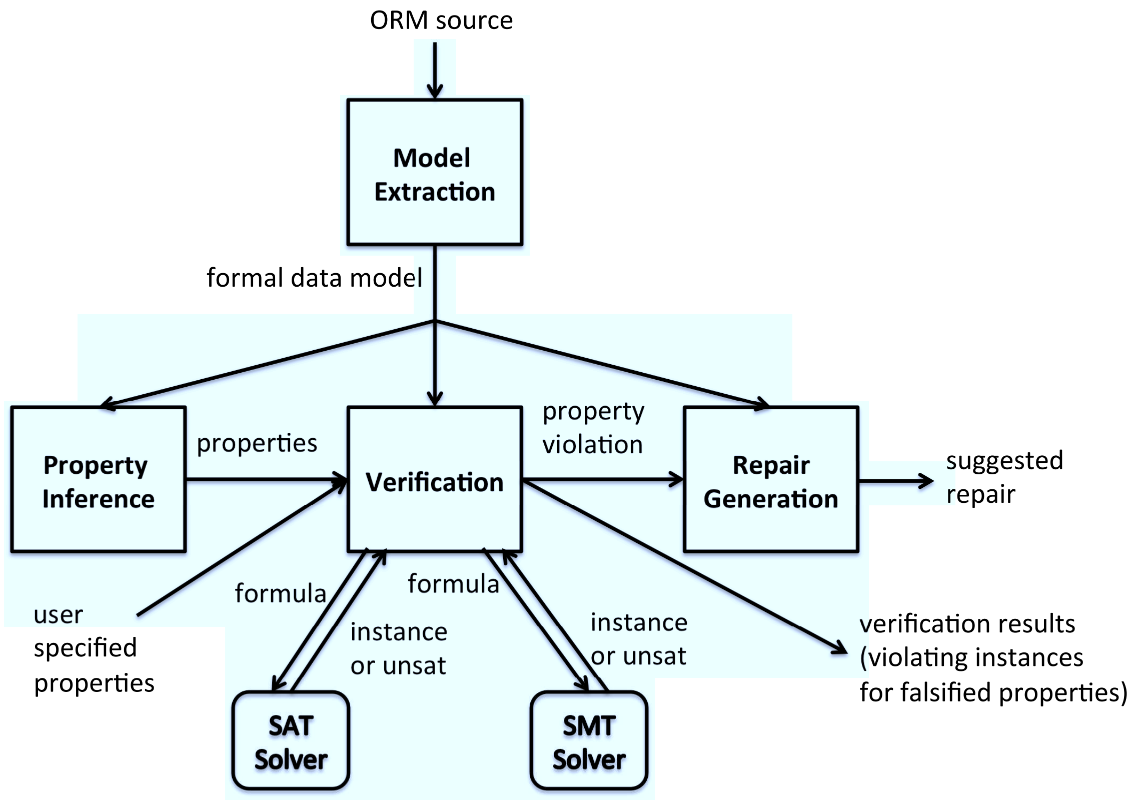


Figure 7.7: Data model analysis toolset.

7.4 Property Repair

In order to check the properties generated by the property inference algorithms presented in the previous section, we integrate the automated verification techniques from the work described in the previous chapters into our tool. The architecture of the resulting tool is shown in Figure 7.7.

Besides automatically inferring properties using the heuristics described in the previous section and verifying them for the Ruby on Rails framework, our tool automatically generates data model repairs for the failed properties. These repairs show how the Rails data model can be modified so that the failed property will hold in the repaired model. The repair rules we developed are discussed below. To clarify the discussion, we will use the Active Records specification in Figure 7.1 to show sample repairs that our tool outputs.

7.4.1 Delete Propagation

First we explain the repair generation for the delete propagation property. If *deletePropagates*(r) fails for some relation $r = (o, t, n, o')$, this means that the data model is set up such that deleting an object of class o will not cause associated objects of o' to be deleted. In order to enforce this property in the data model, the `:dependent` option must be set on the `has_many` or `has_one` declaration corresponding to relation r in o's model. For example, when we run the Inference Algorithm for Delete Propagation (Algorithm 2) on the data model of the application given in Figure 7.1, the *deletePropagates* property is generated for the relation between the User and Context classes. However, this property fails when we check it using the automated verification techniques discussed in the previous section. This means that when a user is deleted, the contexts created by the user are not deleted. In order to enforce this in the data model, the repair our tool generates sets the `:dependent` option on the relation with Context in the User model, i.e.

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

This will cause the deletion of User objects to be propagated to the associated Context objects. Note that the `:dependent` option is set to `:destroy` and not `:delete` since we want the delete to propagate to o''s associated objects. Otherwise there may be objects of another class with a dangling reference to the deleted associated object. In the running example, we observe that setting the `:dependent` option to `:delete` may result in Todo objects with a dangling reference to a deleted Context object. In order to prevent this inconsistency, the `:dependent` option is set to `:destroy` so that the Context model can propagate the delete to the desired relations. Figure 7.8 displays the data model of the todo list application with the automatically generated repair on line 4.

7.4.2 Orphan Prevention

Next we present the repair for the orphan prevention property. When $noOrphans(r)$ fails for a relation $r = (o, t, n, o')$, this mean that the data model is set up such that deleting an object of class o' will cause objects in class o to be orphaned, i.e.

there will be objects of class o that will not be related to any other object. We can enforce this property in the data model by generating a repair that will delete the associated objects that would otherwise be orphaned. This is done by setting the `:dependent` option on the declaration corresponding to relation r in the model for o'. For orphan chains this is repeated down the chain, creating repairs for the declarations that associate a class with the next class in the chain.

For example, when we run the Inference Algorithm for Orphan Prevention (Algorithm 3) on the data model in Figure 7.1, a *noOrphans* property is generated which states that when a User is deleted, no Preference objects should be orphaned. This property fails when we check it using automated verification, which means that when a User is deleted who has a Preference, the Preference object is orphaned. In order to enforce this property in the data model, a repair is generated that sets the `:dependent` on the relation with Preference in the User model, i.e.

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

This will cause the deletion of a User object to be propagated to the associated Preference. There are no more objects in this orphan chain so no further repairs will be generated. This suggested repair, as applied to the data model in Figure 7.1, is shown in Figure 7.8 on line 3.

7.4.3 Transitive Relations

Finally, we discuss the repair for failing transitive relations properties. When *transitive*$(r_0, \ldots, r_m)$ fails for some set of relations $r_0, \ldots, r_m$ it means that r_m is not the composition of the other m relations, as asserted in the property. To repair this property in the data model, we set the `:through` option on the declaration corresponding to the relation $r_m = (o_m, t_m, n_m, o'_m)$ in o_m's data model. For instance, running the Inference Algorithm for Transitive Relations (Algorithm 4) on the example in Figure 7.1 infers the following *transitive* property: the relation between User and Todo should be the composition of the relations between User and Context, and Context and Todo. However, we again find out that this property fails using automated verification. In other words, the todos in the contexts created by a user may not be the same as the todos created by that user. In order to enforce this transitivity in the data model, a repair is generated which sets the `:through` option on the declaration in the User class that associates it with Todo:

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

We also need to remove the `belongs_to :user` declaration in the Todo class since it becomes unnecessary when using the `:through` option. After this repair the relation between User and Todo will be the same as navigating the User-Context relation and then the Context-Todo relation. The data model for the todo list application after being modified by this repair is shown in Figure 7.8 (see lines 5 and 16).

There are two complications in the repair generation of the transitive relation property. For transitive properties with exactly three parameters, $\mathit{transitive}(r_0, r_1, r_2)$, 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 $\mathit{transitive}(r_0, \ldots, r_m)$ 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}.

7.5 Experiments

Our techniques are applicable to ORMs in general, and we have implemented the approach for the Ruby on Rails framework. The property inference techniques

```
class User < ActiveRecord::Base
     has_one :preference, :conditions => "is_active=true",
            :dependent => :destroy
     has_many :contexts, :dependent => :destroy
     has_many :todos, :through => :contexts
end
class Preference < ActiveRecord::Base
     belongs_to :user
end
class Context < ActiveRecord::Base
     belongs_to :user
     has_many :todos, :dependent => :delete
end
class Todo < ActiveRecord::Base
     belongs_to :context
     # line deleted
     has_and_belongs_to_many :tags
end
class Tag < ActiveRecord::Base
     has_and_belongs_to_many :todos
end
```

Figure 7.8: The data model from Figure 7.1 updated with the suggested repairs (in bold) generated by our tool.

were combined with our data model verification tool to provide an end-to-end automated tool for data model verification.

The techniques have been implemented as a toolset for data model analysis, verification and repair. The architecture of the toolset is shown in Figure 7.7. The front end automatically extracts a formal data model from the ORM specification of the web application. The model extraction, property inference, verification and repair components are all integrated together and use the results from the prior stages to generate the results needed for the following stages of the analysis.

We used our toolset to analyze five open source Ruby on Rails applications. Our tool applies our property inference heuristics to the Active Record files of the input web application. The properties inferred are sent to the next component of the tool, which automatically translates the Active Record files to SMT-LIB and then performs verification using the SMT-solver Z3 [100]. If any properties time out during verification (with a time out limit of five minutes), bounded verification is performed instead, using the Alloy Analyzer with a bound of 10, meaning at most 10 objects of each type are instantiated to check satisfaction of these properties.

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

The Applications The five applications used in the experiments are listed in Table 6.1, along with their sizes in terms of lines of code, number of total classes, and number of data model classes. Descriptions of the applications are given below:

- LovdByLess (lovdbyless.com) is a social networking application with the usual features.
- Tracks (getontracks.org) is an application that helps users manage to-do lists, which are organized by contexts and projects.

- OpenSourceRails(OSR) (opensourcerails.com) is a project gallery that allows users to submit, bookmark, and rate projects.
- FatFreeCRM (fatfreecrm.com) is a light-weight customer relations management software.
- Substruct (code.google.com/p/substruct) is an e-commerce application.

Inference and Verification Results The results of running our tool on the five applications are given in Table 7.1. For each application and type of property, it displays the number of properties that were inferred by the tool, the number that failed during verification, and the number that timed out during unbounded verification. A total of 145 properties were inferred, of which 93 failed and 3 timed out during unbounded verification. The three properties that timed out were shown to fail using bounded verification, giving a total of 96 failing properties. We manually investigated each of the failing properties to determine which correspond to data model errors. These results are also summarized in Table 7.1.

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

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

Properties that fail verification are not necessarily errors. For example, a *transitive* property that failed was for LovdByLess, which has forums in which users are allowed to create topics and post messages inside the forum topics. The property inferred states that the relation between User and ForumPost is the transitive between the relations between User and ForumTopic, and ForumTopic and ForumPost. Manual analysis shows that 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.

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

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

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

Performance Our experiments included taking performance measurements as an additional indicator of the effectiveness of our approach. Specifically, we measured the time it took for the inference and verification of each property, as well as the formula size produced by the verification tools. These values were averaged over the properties for each property type per application. The results are summarized in Table 7.2. For the unbounded tool the formula size measures the SMT-LIB specification produced for the property. Here, the number of variables are the number of sorts, functions and quantified variables in the specification, and the number of clauses are the number of asserts, quantifiers and operations. For the bounded tool, the formula size reports the number of clauses and variables created by Alloy's SAT translation. The time taken for repair generation is not reported in Table 7.2 since it is almost zero for all properties.

In summary we see that, for the properties that did not time out during unbounded verification, the longest inference time was 0.22 seconds on average, with the exception of transitive properties for the LovdByLess application which had an average inference time of 1.5 seconds. The longest verification time was only 0.65 seconds on average. Even for the properties that timed out and bounded verification was used instead, we see that figures are reasonable with 2.4 seconds being the longest bounded verification time. What we observe is that inference time is very fast, as is unbounded verification time. Even in the cases for which bounded verification was required, the verification time is very reasonable. In summary, our approach is not only able to find errors effectively, it does so efficiently.

Table 7.1: Inference and Verification Results

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

Table 7.2: Inference and Verification Performance

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

Chapter 8

Integrated Data Model Verifier (iDaVer)

Chapter 7 presented one way to ease the user's burden of manually specifying properties – by inferring properties automatically. However, this approach is limited by the types of properties that can be automatically inferred generally for all applications. To address this issue, we created a tool that accepts *property templates* which facilitate writing application-specific data model properties.

To verify these properties, our tool iDaVer [71] gives the user the option of either bounded or unbounded verification (the approaches described in Chapters 4 and 5, respectively). By integrating these two approaches and offering language-neutral templates to specify properties, iDaVer allows the user to easily switch between the two verification methods, providing a seamless user experience.

This chapter describes property templates and details iDaVer's architecture.

8.1 Overview

iDaVer targets web applications developed using the Rails framework. The front-end of our tool automatically extracts a formal data model from the ORM specification of the input application. Although the formal data model is extracted automatically, the user still has to specify the properties she desires to check about the data model. To facilitate this process, we developed a set of property templates. These templates characterize the most common properties we observed in our earlier research on data model verification (see Chapter 6). These templates can easily be instantiated for different classes and relations by the user.

Our tool verifies properties (specified using property templates) on the automatically extracted formal data model by translating verification queries to satisfiability queries in a specified theory and then using a backend solver for that theory. Our tool combines two different variants of this framework: the SAT-based bounded verification and Satisfiability Modulo Theories (SMT)-based unbounded verification. Our tool integrates these two approaches and provides a unified framework for the verification of Rails data models.

8.2 Property Templates

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

We identified seven property templates that characterize the most common properties we observed in our earlier research on data model verification [72, 73].

```
class User < ActiveRecord::Base
        has_and_belongs_to_many :roles
        has_one :profile, :dependent => :destroy
        has_many :photos, :through => :profile
end
class Role < ActiveRecord::Base
        has_and_belongs_to_many :users
end
class Profile < ActiveRecord::Base
        belongs_to :user
        has_many :photos, :dependent => :destroy
        has_many :videos, :dependent => :destroy,
                          :conditions => "format='mp4'"
end
class Photo < ActiveRecord::Base
        belongs_to :profile
        has_many :tags, :as => :taggable
end
class Video < ActiveRecord::Base
        belongs_to :profile
        has_many :tags, :as => :taggable
end
class Tag < ActiveRecord::Base
        belongs_to :taggable, :polymorphic => true
end
```

Figure 8.1: A data model example

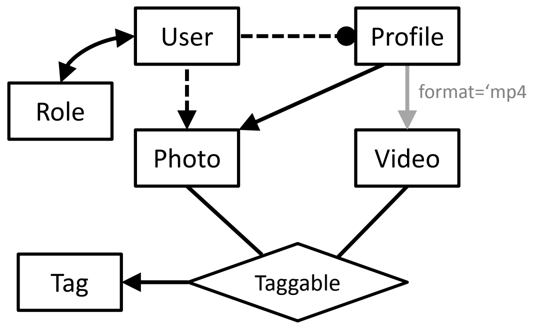


Figure 8.2: The schema extracted from the data model in Figure 8.1.

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. As a running example we use the data model given in Figure 8.1. It is a simplified data model of 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. A graphical representation of the schema extracted for this data model is shown in Figure 8.2.

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 from one class are always related to some object of another class. We formally define this template as

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

For example we can express the following property on the data model in Figure 8.1: *alwaysRelated*(Profile-User). This is saying that a Profile object should always be associated with a User object.

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

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

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

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

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

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

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

$$\textit{transitive}(r_{A-B}, r_{B-C}, r_{A-C}) \equiv$$

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

For the running example, the property *transitive*(User-Profile, 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 a class, o_B, has only one relation, i.e. it is connected to the schema graph via exactly one relation, r_{A-B}. In this case, when an element of class o_A is deleted it is possible that its associated elements in o_B may be *orphaned*—left without any connections to other objects. Formally,

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

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

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

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

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

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,

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

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

8.3 iDaVer's Architecture

The architecture of iDaVer (Integrated DAta model VERifier) is shown in Figure 8.3. iDaVer takes as input a set of model files from a Ruby on Rails 2.0 application and a set of properties in the form of templates. The user can choose one of two verification options: 1) bounded verification with Alloy Analyzer 4, or 2) unbounded verification with the SMT Solver Z3 4.3.

iDaVer extracts a formal data model (discussed in Chapter 3) from the Rails data model files, which is then translated into a specification in the language of the chosen solver. The properties specified (using templates) are also translated

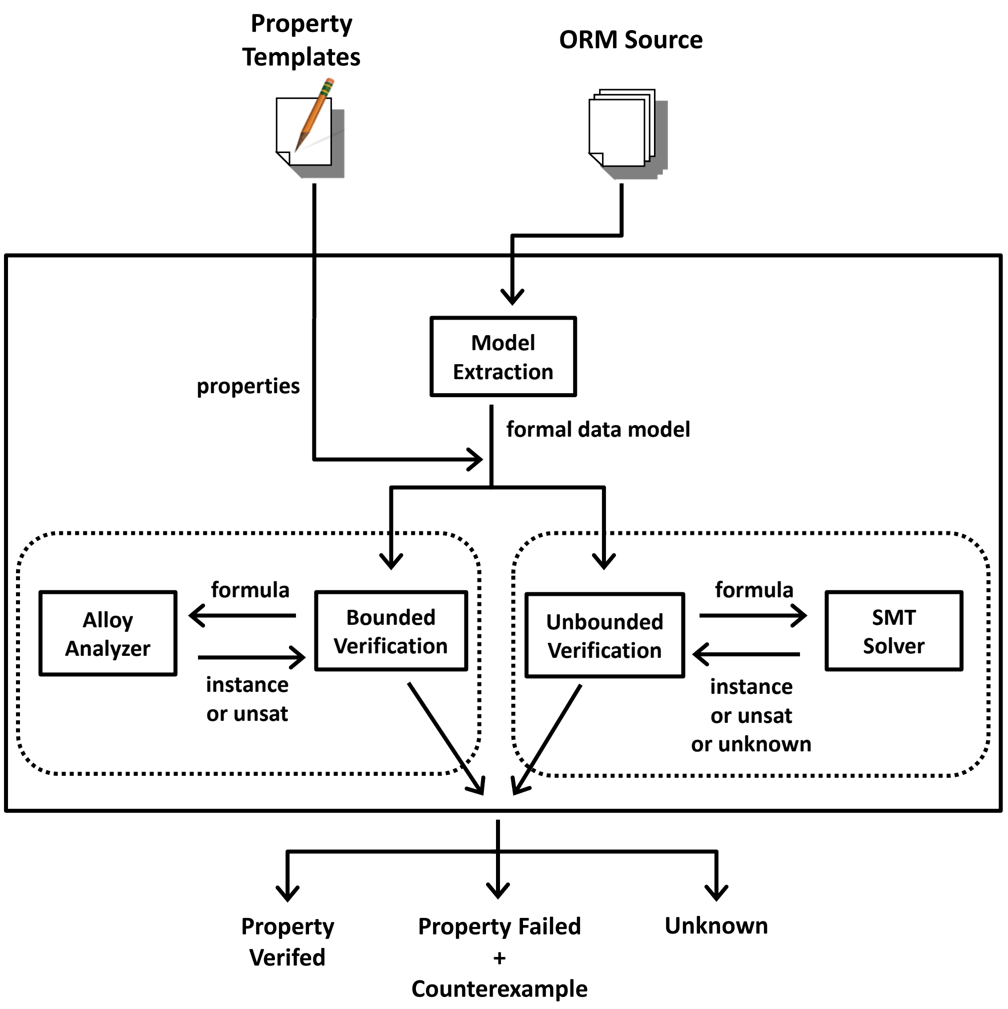


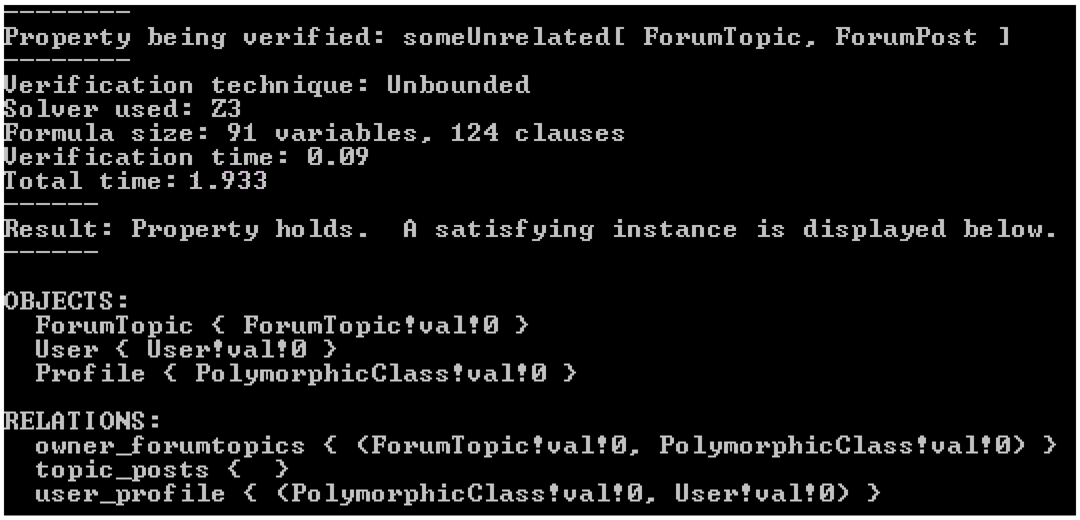
Figure 8.3: Data model analysis toolset.

into the modeling language of the chosen solver and appended to the specification. Then the specification is fed into the solver, and the output of the solver is interpreted by iDaVer and shown back to the user. In cases where the output contains a satisfying or violating instance, iDaVer translates the output of the solver to an instance of the data model (in terms of sets and relations of the data model) before presenting it to the user.

8.4 A Case Study

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

To start, let us say the user desires to verify the data model of the LovdByLess application by ensuring the property `someUnrelated[ForumTopic, ForumPost]` holds, meaning we are checking that it is possible to have topics in a forum that do not have any posts. The user chooses to perform unbounded verification using the SMT solver, Z3. Running iDaVer on these inputs gives the following result[1]:



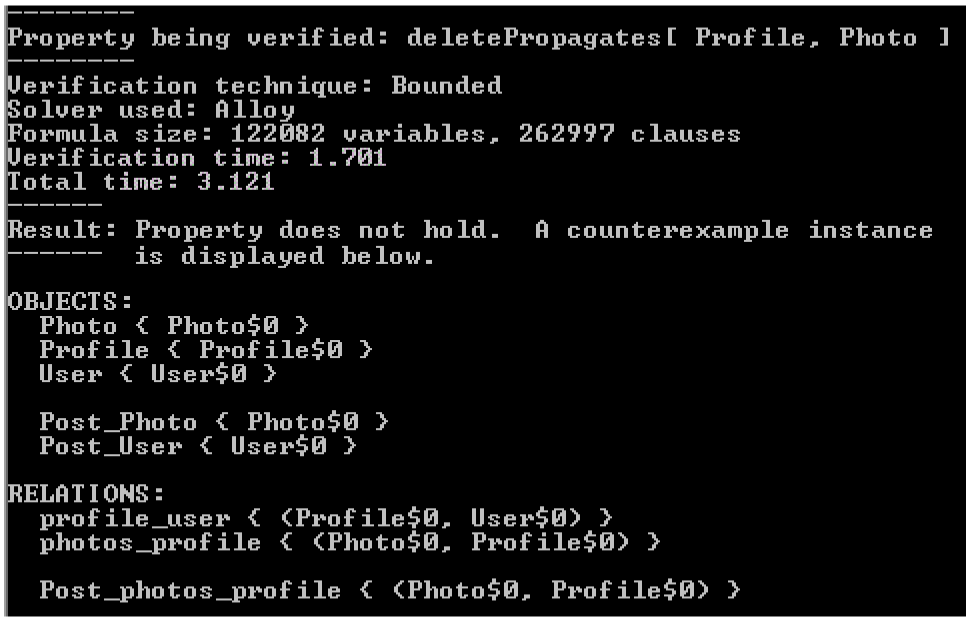
```
---------
Property being verified: someUnrelated[ ForumTopic, ForumPost ]
--------
Verification technique: Unbounded
Solver used: Z3
Formula size: 91 variables, 124 clauses
Verification time: 0.09
Total time: 1.933
------
Result: Property holds.  A satisfying instance is displayed below.
------

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

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

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

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

Finally, iDaVer outputs the result of the verification, which in this case is that the property holds on the data model. Furthermore, iDaVer formulates a satisfying instance from Z3's output, and displays it as object sets and related tuples.

In addition to unbounded verification with Z3, iDaVer can also perform bounded verification using the Alloy Analyzer. Specifying bounded verification and using the default bound of twenty (meaning up to twenty objects of each class will be instantiated to check the property), we input the property `deletePropagates[Profile, Photo]` to check that when a Profile object is deleted, all associated Photo objects will also be deleted. iDaVer gives the following output[1]:



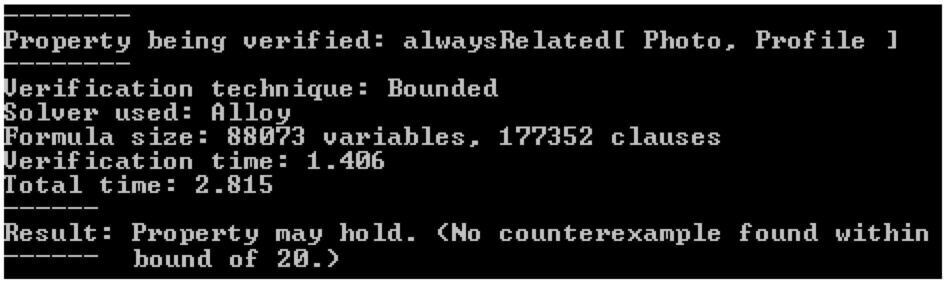
```
--------
Property being verified: deletePropagates[ Profile, Photo ]
--------
Verification technique: Bounded
Solver used: Alloy
Formula size: 122082 variables, 262997 clauses
Verification time: 1.701
Total time: 3.121
------
Result: Property does not hold.  A counterexample instance
------  is displayed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

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

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

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

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

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

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

Even though we performed bounded verification for this property, we were able to determine that the property failed. However this may not always be the case. In fact, the main disadvantage of bounded verification is that its results are not sound for verified assertions and failing predicates. For example, checking that a

Photo object is always associated with some Profile object by running iDaVer with the input `alwaysRelated[Photo, Profile]` and Alloy Analyzer as the verifier, gives the following result:



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

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

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

Let us look at one final example. To check that deleting a Blog entry does not cause any Comment objects to have a dangling reference, we check the `noDangling[Blog, Comment]` property. Using Z3 to do unbounded verification, iDaVer returns that this property does not hold on the data model and provides a counterexample to help the user pinpoint the error. When the user runs the application and deletes a Blog entry, we see that in the database the associated Comments are left with a dangling reference to the deleted Blog entry. In the application, the user's main page contains a list of recent activity. This includes when someone has commented on that user's blog entry. The application sees there is a comment made for this user, but it cannot find the referenced blog entry. The application tries to make up for this error by returning an empty string. An application error occurs nonetheless since this empty string is displayed on the screen where text is expected.

Using iDaVer, the user discovers a bug that can now be fixed in the data model by setting the `:dependent` option on the relation to Comment in the Blog class, so that all Comments are deleted when a Blog entry is deleted. This example demonstrates the importance of verifying data models and using the data modeling constructs available in Rails to enforce properties of data models. Using other parts of the application to enforce properties that should and can be upheld by the data model may lead to errors, such as this one.

Chapter 9

Related Work

In this chapter we present related work. We discuss work on web application modeling and verification, and work on data model analysis and verification. We also present works related to the property-based data model projection technique presented in Chapter 5, and the automated property inference techniques presented in Chapter 7.

9.1 Modeling and Verification of Web Applications

Formal modeling and automated verification of web applications has been investigated before. There has been some work on analyzing navigation behavior in web applications. It focuses on the correct handling of 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 [56]. For example, the problem of navigation inconsistencies in web applications has been studied [62], where it has been shown that multiple browser windows can lead the user of a popular travel reservation site to purchase the wrong flight.

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 [90] and investigated further later on [49, 9, 46, 61]. State machine based models have been used to automatically generate test sequences [99], perform some form of model checking [84, 48] and for runtime enforcement [46]. In contrast to these previous efforts, we are focusing on analysis of the data model rather than the navigational aspects of web applications. Some language based solutions have been proposed to alleviate this problem, in which such navigation inconsistencies reduce to type checking errors [42]. Also, Alloy has been used for discovering bugs in web applications related to browser and business logic interactions [10]; this is yet another work verifying the navigational aspects of web applications as opposed to the data model as in our work.

There has been some prior work on formal modeling of web applications using the Unified Modeling Language (UML) [17] and extending UML to capture complex web application behavior such as browsing and operations on navigation states [5]. WebML [13] is a modeling language developed specifically for modeling web applications. Formal specification of input validation rules for web applications, where an implementation can be generated from the formal specification has been investigated [8]. These efforts focus on model-driven development whereas our approach is a reverse engineering approach that extracts the model of an already-existing application and analyzes it to find errors.

There has been some work on semi-automatically extracting a UML model from a web application [92], but its goal is to test for design flaws. In [92], analysis is performed on the extracted model, although they just focus on analyzing usage statistics to aid in testing of the web application. There has a been a lot of work done on automated testing of web applications [29, 37, 3, 4, 15, 83, 89, 59]. The closest related work in this domain is for generating test cases for databases [37]. In this work, test cases are automatically synthesized and are composed of a sequence of HTTP requests that simulate a user session which will obtain a certain database state. This technique is unimplemented but can be used to discover workflow attacks, whereas the goal of our approach is to discover data modeling errors. In contrast to all these works, we perform static verification of data models rather than testing.

The verification of traditional web applications, which do not use object-relational mappings, has also been investigated in recent years [51, 25, 36, 30, 58, 27, 26, 28]. The techniques presented in [51, 30, 58] are reverse-engineering approaches like ours. However, they focus on the verification of navigation properties whereas we check for data modeling errors. The approach in [58] automatically extracts the page transitions of a Struts/Java Server Pages web application and translates them into a Promela specification. Then the model checker SPIN [52] is used for verification. The work described in [30] automatically translates the UML design of a web application into an SMV model and uses the SMV model checker [68] to verify navigation properties. In [51] a slightly different approach is taken by using dynamic analysis to extract the formal model: a finite automata model is created from a recorded browser session. This model is then verified using SPIN. All these approaches, like ours, require properties to be manually written and fed to the model checker to perform verification.

A different approach to the verification of web applications is presented in [25]. It requires manual annotations to the code, but does not require properties to be written. In this work, annotations in JML [12] are used to specify how components interact with a shared session data repository. Then static verification is used to ensure each component implementation obeys its contract specification. Besides performing static verification of a global client-server interaction protocol and a runtime policy enforcement, their tool guarantees that there will be no broken data dependencies.[1] In contrast, the work in [36] is fully automated, in that dynamic analysis is used to observe the normal operation of a web application to infer likely invariants. Then symbolic model checking is performed (using an extended version of JPF [50]) over symbolic input to identify program paths likely to violate those invariants. This approach discovers logic vulnerabilities in web applications.

Finally, there are previous results on unbounded verification of data-driven web applications based on manually written high level specifications [27, 26, 28]. The work in [26] formally defines a data-driven web application and provides theoretical results about the decidability of a class of navigation properties. In [27, 28] these results are implemented in a verification tool for data-driven web applications which are specified using a high-level language such as WebML.

In our dissertation, by focusing on three-tier style web applications we are able to exploit the modularity in the three-tier architecture and extract formal

[1]Broken data dependencies happen when a client request causes a data item to be read from the server side shared session repository before it has actually been written, or for a shared data read interaction when the object returned by the read is not of the type expected.

data models from existing applications without requiring any manual modeling or specification.

A very related work is Rubicon [69], a bounded verification approach for Ruby on Rails web applications. In this work, behavioral bugs are discovered by first manually converting RSpec [16] tests into the Rubicon specification language, and then using symbolic execution in conjunction with the Alloy Analyzer to perform bounded verification. In comparison to our work, Rubicon verifies the Controller aspect of an application as opposed to the Data Model. Another difference in the approach is that Rubicon requires the specification of application behavior rather than properties in order to perform verification, as in our work. We also give the option of unbounded verification in addition to bounded verification.

9.2 Data Model Analysis and Verification

There has been some prior work on using Alloy for data model analysis. For example, mapping relational database schemas to Alloy has been studied before [20]. 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 [95]. However, unlike our work, the translation to Alloy is not automated in these earlier efforts.

Formal specification of access control policies in conjunction with a data model using Alloy has also been studied, where an implementation is automatically synthesized from the formal specification [14]. There has also been work done on automatically translating Alloy specifications to a database schema with integrity constraints [57]. The integrity constraints are upheld by automatic repair of the database after each update operation. These efforts follow the model-driven development approach whereas our approach is a reverse engineering approach that extracts the model of an existing application and analyzes it to find errors.

There has been recent work on the specification and analysis of conceptual data models [87, 67, 47]. These works focus on checking the consistency of models (i.e. checking that an instance exists which satisfies all constraints in the model), as well as on efficient generation of satisfiable instances as test data. These efforts focus on model-driven development and testing whereas our approach is a reverse-engineering approach for the verification of data models.

There have been efforts in formal specification, verification and analysis of object-oriented models. For example, the Object Constraint Language (OCL) [76, 96] is a specification language for describing constraints on object-oriented models. OCL is primarily used for specifying class invariants on fields and associations, and for specifying pre and post conditions of class methods. Being one of the components of the UML [76], OCL is a commonly used formal language for object-oriented modeling, especially for expressing precise constraints that cannot be expressed using UML diagrams. OCL can also be used to specify integrity constraints of relational databases, as shown in [24]. Research on verification of OCL specifications have ranged from simulation of object-oriented models [81], to interactive verification via automated theorem prover support [1].

One attempt to check the correctness of UML/OCL models is the UML-based Specification Environment (USE) [81, 39]. USE provides an environment where users can simulate the behavior of UML models and check OCL invariants and pre and post conditions during simulation. One disadvantage of USE is the lack of support for automatically guided simulation, and hence one can only cover a small portion of system behaviors with USE. Another approach for the validation of UML models constrained by OCL constraints is described in [66, 2, 65]. They each propose a translation from UML class diagrams to Alloy. This allows one to use the Alloy Analyzer for simulation as well as bounded model checking. The techniques presented in [2, 65] is implemented whereas the approach in [66] is not. The approach in [2] has an extended support of UML features compared to [65]. In contrast, to these types of simulation-based validation, we applied unbounded, automated verification techniques to guarantee the correctness of data models.

Note that, both UML/OCL and Alloy require manual specification of the object-oriented data models. In this dissertation, we automatically extract formal data models from existing applications without requiring manual intervention.

Alloy and UML/OCL are two object-oriented modeling languages; there are there are other modeling languages such as the popular Z [45] and B [60] languages. Like Alloy and OCL, the predominant data types in Z and B are sets and relations. However, both Z and B were designed with proving in mind rather than lightweight analysis like Alloy was. Thus both languages have substantial support of specialized theorem provers. Furthermore, both support higher-order structures, whereas Alloy is strictly first-order; thus Alloy is less expressive and has fewer constructs. B is more like an abstract programming language rather than a specification language; in fact, a design goal of B was to produce code from abstract models. This make B less flexible than the other languages. Z, like Alloy, is essentially a logic with extra constructs added for easy expression of software abstractions. Both Z and B describe behavior using constraints, as do Alloy and OCL. In our work, Alloy was chosen due to its simplicity and the automatic analysis capability provided by the Alloy Analyzer. For the unbounded analysis portion of our work we chose SMT-LIB so we could perform verification using SMT solvers and their ability to produce counter-examples.

There has been some recent work on unbounded verification of Alloy specifications using SMT solvers [38], but to the best of our knowledge this approach has not been implemented yet. Unbounded verification of Alloy specifications may be more challenging than the data model verification problem that we focus on in this dissertation since the Alloy language provides powerful constructs such as transitive closure.

The fact that verification with SMT solvers can be more efficient than SAT-based bounded verification has been observed in other verification domains [19]. However, the data model verification problem we investigate in this dissertation is different from the problems studied in these earlier works.

Another category of related work is on the verification of abstract data types [44, 97]. Verification is done using a technique called data type induction. This is analogous to the way we verified behavior properties in our approach. In data type induction, induction is performed on the state of the abstract data type resulting from applying any of the defined operations. In our approach, we applied this idea to the data model operation we modeled – the delete object operation.

9.3 Dependency-Based Projection

In Chapter 5 we presented our property-based data model projection algorithm. There has been earlier work on reducing the cost of automated verification. These techniques are based on the concept of program slicing, which was first introduced by Weiser [98] and extended in various ways, such as [78, 7]. The intended use of program slicing was to obtain an executable program with less statements than the original. Slicing is performed based on the slicing criterion–a set of variables and a program point–and statements are removed if they don't have a direct or indirect effect on the values of the criterion. The slicing algorithm is based on data flow and control flow dependences. Thus these techniques are based on the ordering of program statements and the semantics of the programming language, whereas our algorithm is based on the semantics of data models. Further, our goal is different from these works: to reduce the cost of verification as opposed to aid in debugging.

Reducing the cost of verification in the domain of program slicing is seen in [6, 18]. [6] presents several techniques for compositional reasoning in model checking. The most related technique is the cone of influence reduction, which simplifies program processes based on a set of shared variables in order to preserve the properties that refer to those variables whilst reducing the size of the system.

[18] presents Bandera, a toolset to verify multi-threaded Java programs. It implements a property-based slicing technique that extends the traditional data and control dependency method to include inter-thread synchronization dependences. These works present program analysis-based reductions whereas our domain is data model simplification.

Program slicing techniques have also been applied to declarative models, including Alloy specifications [94]. Slicing UML/OCL class diagrams based on the property to be verified has also been studied [86, 85]. The slicing procedure described in [86, 85] not only removes irrelevant information, but also breaks the original model into submodels to verify them independently. The partitioning criteria includes relationship dependences, which is similar to our approach (although we do not fragment our model). Compared to these earlier results, our projection algorithm is a specialized reduction technique for data model verification that utilizes the data model semantics.

9.4 Automated Property Inference and Repair

Automated discovery of likely program invariants by observing runtime behaviors of programs has been studied extensively [32, 33, 34]. There has also been extensions of this style of analysis to inference of abstract data types [43].

Instead of using observations about the runtime behavior of programs, we analyze the static structure of the data model extracted from the ORM specification to infer properties. Static verification of inferred properties has been investigated earlier [75]. 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 [23, 22, 31, 64]. In this dissertation we are not focusing on generating code for fixing data model properties during runtime. Instead, we generate repairs that modify the data model declarations that fix the data model for all possible executions. Moreover, we focus on data model verification in web applications based on ORM specifications which is another distinguishing feature of our work.

Chapter 10

Conclusion

Most modern web applications are built using development frameworks based on the Model-View Controller (MVC) pattern. The MVC pattern facilitates the separation of the data model (Model) from the user interface logic (View) and the control flow (Controller). The data model specifies the types of objects used by the application and the relations among them. Since the data model forms the foundation of such applications, its correctness is crucial.

In this dissertation, we presented our work on increasing the dependability of data models. To this end, we developed two data model verification approaches. The first is a bounded verification technique. We automatically extract a formal data model from the data model specification and then translate verification queries about these models into the Alloy language. Finally, a boolean SAT solver in the form of Alloy Analyzer is used to check the satisfiability of the resulting formulas.

The second approach is an unbounded verification technique for data models. We once again exploit the inherent modularity in MVC frameworks to automatically extract a formal data model from the data model specification. Next, verification queries about these models are automatically translated to satisfiability queries in the theory of uninterpreted functions. We then use an SMT solver to check the satisfiability of the resulting formulas.

We integrated these two approaches and implemented them in a tool called iDaVer, a verifier of data models of web applications written using the Ruby on Rails framework. Since the verification techniques required the user to manually write the properties they desire to verify about the data model, iDaVer has the added feature of property templates. Property templates simplify the task of property specification and allows iDaVer to be applied directly on application code, not requiring the user to be familiar with any specialized modeling language or formal notation. We applied iDaVer to five open-source applications. Our results demonstrate that the proposed approaches are feasible, with different tradeoffs between the bounded and unbounded verification approaches.

Finally, in this dissertation we presented techniques for automatic property inference and repair for data models. This was done by first extracting the formal data model schema of a given application. Then the structure of the relations in the schema are analyzed to infer three types of properties. This automatic property inference techniques can be combined with data model verification techniques, such as the ones we developed. We implemented such a tool to check if the inferred properties hold on the data model. For failing properties it generates 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.

Bibliography

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hahnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The key tool. *Software and Systems Modeling*, 4(1):32–54, 2005.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. Uml2alloy: A challenging model transformation. In *Proc. Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, LNCS 4735*, pages 436–450, 2007.

[3] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4:326–345, 2005.

[4] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, 2011.

[5] L. Baresi, F. Garzotto, and P. Paolini. Extending UML for modeling web applications. In *Proc. 34th Ann. Hawaii Int. Conf. Sys. Sci. (HICSS)*, 2001.

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

[7] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[8] M. Book, T. Brückmann, V. Gruhn, and M. Hülder. Specification and control of interface responses to user input in rich internet applications. In *Proc. 24th Int. Conf. on Automated Software Engineering (ASE)*, pages 321–331, 2009.

[9] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *Proc. 19th Int. Conf. Automated Software Engineering (ASE)*, pages 100–109, 2004.

[10] B. Bordbar and K. Anastasakis. MDA and analysis of web applications. In *Proc. VLDB Workshop on Trends in Enterprise Application Architecture*, pages 44–55, 2005.

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

[12] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[13] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (WebML): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[14] F. Chang. *Generation of Policy-rich Websites from Declarative Models.* PhD thesis, MIT, 2009.

[15] W.-K. Chang, S.-K. Hon, and C.-C. W. Chu. A systematic framework for evaluating hyperlink validity in web environments. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, pages 178–185, 2003.

[16] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North. *The RSpec Book.* The Pragmatic Bookshelf, Dallas, 2010.

[17] J. Conallen. Modeling web application architectures with UML. *Commun. ACM*, 42(10):63–70, 1999.

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

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

[20] A. Cunha and H. Pacheco. Mapping between Alloy specifications and database implementations. In *Proc. 7th Int. Conf. Engineering and Formal Methods (SEFM)*, pages 285–294, 2009.

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

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

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

[24] B. Demuth and H. Hußmann. Using uml/ocl constraints for relational database design. In *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, pages 598–613, 1999.

[25] L. Desmet, P. Verbaeten, W. Joosen, and F. Piessens. Provable protection against web application vulnerabilities related to session data dependencies. *IEEE Trans. Software Eng.*, 34(1):50–64, 2008.

[26] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73(3):442–474, 2007.

[27] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. A system for specification and verification of interactive, data-driven web applications. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 772–774, 2006.

[28] A. Deutsch and V. Vianu. Wave: Automatic verification of data-driven web services. *IEEE Data Eng. Bull.*, 31(3):35–39, 2008.

[29] G. Di Lucca, A. Fasolino, F. Faralli, and U. De Carlini. Testing web applications. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 310 – 319, 2002.

[30] F. M. Donini, M. Mongiello, M. Ruta, and R. Totaro. A model checking-based method for verifying web application design. *Electr. Notes Theor. Comput. Sci.*, 151(2):19–32, 2006.

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

[32] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. 1999 Int. Conf. Software Engineering (ICSE)*, pages 213–224, 1999.

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

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

[35] Fat free crm. http://www.fatfreecrm.com/.

[36] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. *Toward automated detection of logic vulnerabilities in web applications*. University of California, Santa Barbara, 2010.

[37] X. Fu. Relational constraint driven test case synthesis for web applications. In *Proceedings Fourth International Workshop on Testing, Analysis and Verification of Web Software, EPTCS 35*, pages 39–50, 2010.

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

[39] M. Gogolla, J. Bohling, and M. Richters. Validation of uml and ocl models by automatic snapshot generation. In *Proc. UML 2003, LNCS 2863*, 2003.

[40] Google Health. http://health.google.com/.

[41] Google Powermeter. http://www.google.org/powermeter/.

[42] P. T. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions and errors. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, LNCS 2618*, pages 238–252, 2003.

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

[44] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12):1048–1064, 1978.

[45] A. Hall. Using z as a specification calculus for object-oriented systems. In *Proc. VDM '90, VDM and Z - Formal Methods in Software Development, Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, LNCS 428*, pages 290–318, 1990.

[46] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proc. 25th Int. Conf. Automated Software Engineering (ASE)*, pages 235–244, 2010.

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

[48] M. Han and C. Hofmeister. Modeling and verification of adaptive navigation in web applications. In *Proceedings of the 6th International Conference on Web Engineering, ICWE 2006, Palo Alto, California, USA, July 11-14, 2006*, pages 329–336, 2006.

[49] M. Han and C. Hofmeister. Relating navigation and request routing models in web applications. In *10th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 346–359, 2007.

[50] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:366–381, 2000.

[51] M. Haydar. Formal framework for automated analysis and verification of web-based applications. In *Proc. 19th Int. Conf. Automated Software Engineering (ASE)*, pages 410–413, 2004.

[52] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[53] D. Jackson. A comparison of object modelling notations: Alloy, uml and z. Technical report, 1999.

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

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

[56] S. Krishnamurthi, R. B. Findler, P. Graunke, and M. Felleisen. *Modeling Web Interactions and Errors*, pages 255–275. Springer, 2006.

[57] S. Krishnamurthi, K. Fisler, D. J. Dougherty, and D. Yoo. Alchemy: transmuting base alloy specifications into implementations. In *Proceedings of the*

16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008, pages 158–169, 2008.

[58] A. Kubo, H. Washizaki, and Y. Fukazawa. Automatic extraction and verification of page transitions in aweb application. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 350 –357, 2007.

[59] D. Kung. An agent-based framework for testing web applications. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 174 – 177, 2004.

[60] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag New York, Inc., 1st edition, 1996.

[61] K. R. P. H. Leung, L. C. K. Hui, S.-M. Yiu, and R. W. M. Tang. Modeling web navigation by statechart. In *24th International Computer Software and Applications Conference (COMPSAC 2000), 25-28 October 2000, Taipei, Taiwan*, pages 41–47, 2000.

[62] D. R. Licata and S. Krishnamurthi. Verifying interactive web programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 164–173, 2004.

[63] Lovdbyless. http://lovdbyless.com/.

[64] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *Proc. 24th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, pages 620–624, 2009.

[65] S. Maoz, J. O. Ringert, and B. Rumpe. Cd2alloy: Class diagrams analysis using alloy revisited. In *Proc. Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. LNCS 6981*, pages 592–607, 2011.

[66] T. Massoni, R. Gheyi, and P. Borba. A uml class diagram analyzer. In *In 3rd International Workshop on Critical Systems Development with UML, affiliated with 7th UML Conference*, pages 143–153, 2004.

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

[68] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[69] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 60, 2012.

[70] J. Nijjar, I. Bocic, and T. Bultan. Data model property inference, verification and repair for web applications. (submitted).

[71] J. Nijjar, I. Bocic, and T. Bultan. An integrated data model verifier with property templates. In *Proc. FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 29–35, 2013.

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

[73] J. Nijjar and T. Bultan. Unbounded data model verification using smt solvers. In *Proc. 27th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, pages 210–219, 2012.

[74] J. Nijjar and T. Bultan. Data model property inference and repair. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 202–212, 2013.

[75] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[76] Omg unified modeling language specification, version 1.3. http://www.omg.org.

[77] Open source rails. http://www.opensourcerails.com/.

[78] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 177–184, 1984.

[79] OWASP Top Ten Project. Retrieved May 2007, from http://www.owasp.org/.

[80] Parsetree. http://rubyforge.org/projects/parsetree/.

[81] M. Richters and M. Gogolla. Validating uml models and ocl constraints. In *Proc. UML 2000, LNCS 1939*, 2000.

[82] R. Rivest. S-expressions. Network Working Group, Internet Draft.

[83] S. Sampath, V. Mihaylov, A. L. Souter, and L. L. Pollock. Scalable approach to user-session based testing of web applications through concept analysis. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pages 132–141, 2004.

[84] E. D. Sciascio, F. M. Donini, M. Mongiello, R. Totaro, and D. Castelluccia. Design verification of web applications using symbolic model checking. In *Proc. 5th Int. Conf. Web Engineering (ICWE)*, pages 69–74, 2005.

[85] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon. Verification-driven slicing of uml/ocl models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 185–194, 2010.

[86] A. Shaikh, U. K. Wiil, and N. Memon. Evaluation of tools and slicing techniques for efficient verification of uml/ocl class diagrams. *Adv. Software Engineering*, 2011, 2011.

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

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

[89] S. Sprenkle, E. Gibson, S. Sampath, and L. L. Pollock. Automated replay and failure detection for web applications. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 253–262, 2005.

[90] P. D. Stotts, R. Furuta, and C. R. Cabarrus. Hyperdocuments as automata: Verification of trace-based browsing properties by model checking. *ACM Trans. Inf. Syst.*, 16(1):1–30, 1998.

[91] Substruct. http://code.google.com/p/substruct/.

[92] P. Tonella and F. Ricca. Dynamic model extraction and statistical analysis of web applications. In *WSE*, pages 43–52, 2002.

[93] Tracks. http://getontracks.org/.

[94] E. Uzuncaova and S. Khurshid. Program slicing for declarative models. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–2, 2006.

[95] L. Wang, G. Dobbie, J. Sun, and L. Groves. Validating ORA-SS data models using Alloy. In *17th Australian Software Engineering Conference (ASWEC)*, pages 231–242, 2006.

[96] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[97] B. Wegbreit and J. M. Spitzen. Proving properties of complex data structures. *J. ACM*, 23(2):389–396, 1976.

[98] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, pages 439–449, 1981.

[99] S. Yuen, K. Kato, D. Kato, , and K. Agusa. Web automata: A behavioral model of web applications based on the MVC model. *Information and Media Technologies*, 1(1):66–79, 2006.

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