Bounded Verification of Ruby on Rails Data Models

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

ABSTRACT

The use of scripting languages to build web applications has increased programmer productivity, but at the cost of degrading dependability. In this paper we focus on a class of bugs that appear in web applications that are built based on the Model-View-Controller architecture. Our goal is to automatically discover data model errors in Ruby on Rails applications. To this end, we created an automatic translator that converts data model expressions in Ruby on Rails applications to formal specifications. In particular, our translator takes Active Records specifications (which are used to specify data models in Ruby on Rails applications) as input and generates a data model in Alloy language as output. We then use bounded verification techniques implemented in the Alloy Analyzer to look for errors in these formal data model specifications. We applied our approach to two open source web applications to demonstrate its feasibility.

Categories and Subject Descriptors

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

General Terms

Verification

Keywords

automated verification, bounded verification, MVC frameworks, data model, web application modeling and analysis

1. INTRODUCTION

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 to the applications written using these scripting languages. Current web software development processes rely on manual testing for eliminating bugs.

ISSTA'11, July 17 - 21, 2011, Toronto, ON, Canada

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.

In this paper we present a bounded verification approach for data models of web applications that are written using the Ruby on Rails framework (Rails for short). Rails is a web development framework for the Ruby language, organized around the Model-View-Controller (MVC) architecture [4]. The MVC architecture facilitates the separation of the data model (Model) from the user interface logic (View) and the control flow logic (Controller). Due to the modularity and separation of concerns principles imposed by the MVC architecture, the data model specifications in Rails applications can be separated from the other application logic. This makes it feasible to perform verification on the data model of an application in isolation.

Our verification approach works as follows: We first automatically translate the Rails data models to formal specifications. Then, we write properties about the data model that we expect to hold. Next, we use bounded verification techniques to check if these properties hold on all the instances of the given data model within a given bound. We implemented this approach by writing a translator that translates the Rails data model specifications to Alloy specifications [16]. (Our translator targets Rails version 2 and the Alloy Analyzer version 4). We add the properties about the data model to this automatically generated Alloy specification. We then use the Alloy Analyzer to check these properties. The Alloy Analyzer converts bounded verification queries to Boolean SAT problems and uses a SAT-solver to determine the result.

To evaluate the effectiveness and usability of our approach, we applied it to two open-source Rails applications, TRACKS and FatFreeCRM. The model files of these two applications were fed into our tool to generate formal data model specifications. Using the Alloy Analyzer we found that some properties which could be enforced in the data model were not being enforced in these applications.

The rest of the paper is organized as follows: Section 2 describes the Rails data models. Section 3 formalizes the data model verification problem. Section 4 describes how

```
class Account < ActiveRecord::Base
    belongs_to :user
    has_many :account_contacts, :dependent => :destroy
    has_many :contacts, :through => :account_contacts
    has_one :address, :as => :addressable,
        :dependent => :destroy,
        :conditions => "address_type='Billing'"
end
class AccountContact < ActiveRecord::Base
    belongs_to :account
    belongs_to :contact
end
class Address < ActiveRecord::Base
    belongs_to :addressable, :polymorphic => true
end
class Contact < ActiveRecord::Base
    belongs_to :user
    has_one :account_contact, :dependent => :destroy
    has_one :account, :through => :account_contact
    has_one :address, :as => :addressable,
        :dependent => :destroy
end
class User < ActiveRecord::Base
    has_many :accounts
    has_many :contacts
end
```

Figure 1: Simplified data model for FatFreeCRM

we translate Rails data models to the Alloy language. Section 5 describes our experiments. Section 6 discusses related work, and Section 7 presents our conclusions.

2. RAILS DATA MODELS

In this section, we describe how data models are expressed in Rails applications. The object-relational mapping Rails uses 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 use the running example shown in Figure 1 to describe Rails' data-modeling features. This is a simplified version of the data model from an open source Customer Relations Management software called FatFreeCRM [12]. Customers are typically companies, and for each company an `Account` is created by a `User`. `Users` also create `Contacts`. `Contacts` are people in the companies who serve as the main person of contact for that company. Finally, each `Contact` and `Account` have an `Address`. (There is also an `AccountContact` class which is used to create a many-to-many relationship between `Account` and `Contact`, which we explain below.)

2.1 Three Basic Relationships in Data Models

Active Records support three basic types of relationships: 1) *one-to-one:* An Object A is associated with zero or one Object B's. So, more accurately, this is a one-to-zero-or-one relationship. 2) *one-to-many:* An Object A is associated with an arbitrary (zero or more) number of Object B's. 3) *many-to-many:* An arbitrary (zero or more) number of Object A's are associated with an arbitrary (zero or more) number of Object B's. 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 Rails models of the related objects:

1. one-to-one: Declaring a one-to-one relationship between `Contact` and `AccountContact` objects:

```
class Contact < ActiveRecord::Base
  has_one :account_contact
end
```

```
class AccountContact < ActiveRecord::Base
  belongs_to :contact
end
```

2. one-to-many: Declaring a one-to-many relationship between `User` and `Account` objects:

```
class User < ActiveRecord::Base
  has_many :accounts
end
```

```
class Account < ActiveRecord::Base
  belongs_to :user
end
```

3. many-to-many: Declaring a many-to-many relationship between `Account` and `Contact` objects [1]:

```
class Account < ActiveRecord::Base
  has_and_belongs_to_many :contacts
end
```

```
class Contact < ActiveRecord::Base
  has_and_belongs_to_many :accounts
end
```

Note that in order to express the inheritance relation in Rails the notation `ChildClass < ParentClass` is used. Most objects will inherit from the `ActiveRecord::Base` class, as seen in the examples above. This is so that the data objects inherit all the database-connection functionality that is located in the `ActiveRecord` class.

2.2 Extending the Data Relationships

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

The `:through` *Option.* The first option we want to discuss is the `:through` option for the `has_many` and `has_one` declarations. Let us assume that we are using the `:through` option with the `has_many` declaration. The `:through` option is used when ObjectA has a one-to-many relation with ObjectB, ObjectC also has a one-to-many relation with ObjectB, and the Rails programmer would like direct access from ObjectA to ObjectC. For instance, let us say a data model is set up such that a Book has many Authors, and a Book also has many Editions. To get all the different editions of books an author has worked on, the programmer would have to write code to obtain the set of Books an Author has worked on, and then get the set of Editions of those Books. However, by using the `:through` option, the programmer can declare that Authors have many Editions `:through` Books. This will allow the programmer to directly access the set of book Editions from an Author object. Another use of the `:through` option is for setting up a many-to-many relation using a join model as opposed to a join table using the `has_and_belongs_to_many` declaration.

[1]Although FatFreeCRM does not have a direct many-to-many relationship between `Accounts` and `Contacts`, we present one here for illustrative purposes.

We see this following use of the `:through` option between `Accounts` and `Contacts` in FatFreeCRM.

```
class Account < ActiveRecord::Base
    has_many :account_contacts
    has_many :contacts, :through => :account_contacts
end
class Contact < ActiveRecord::Base
    has_one :account_contact
end
class AccountContact < ActiveRecord::Base
    belongs_to :account
    belongs_to :contact
end
```

The `:conditions` *Option.* The second option that can be used to extend 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 Account < ActiveRecord::Base
    has_one :address,
        :conditions => "address_type='Billing'"
end
class Address < ActiveRecord::Base
    belongs_to :account
end
```

The `:conditions` option limits the relationship to those objects that meet a certain criteria. In this example, `Account` objects are only related to an `Address` object if the `address_type` field is `Billing`. The condition statement needs to be in the form of the WHERE clause of a SQL query.

The `:polymorphic` *Option.* Rails supports declaration of polymorphic associations. This is similar to the idea of interfaces in object oriented design, where we have dissimilar things that have common characteristics which are embodied in the interface they implement. In Rails, polymorphic associations are declared by setting the `:polymorphic` option on the `belongs_to` declaration. We see this setup in FatFreeCRM between `Address`, `Accounts` and `Contacts`. Although accounts and contacts are not similar enough to have a sub-class relationship, both have an address. By using the `:polymorphic` option in the `Address` class we set up an interface that allows any and all classes that have an address to create an association with the `Address` class. Any classes created in the future can also take part in this relationship, all without having to make any changes to the `Address` class. The models for the `Account`, `Contact` and `Address` classes are given below.

```
class Address < ActiveRecord::Base
    belongs_to :addressable, :polymorphic => true
end
class Account < ActiveRecord::Base
    has_one :address, :as => :addressable
end
class Contact < ActiveRecord::Base
    has_one :address, :as => :addressable
end
```

The polymorphic relationship is expressed in `Account` and `Contact` using the `has_one` declaration with an `:as` option. (The `:as` option can also be specified on a `has_many` declaration.)

The `:dependent` *Option.* The final Rails construct we want to 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 `has_many`), deleting this object will also delete the 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 dependences, whereas `:destroy` first checks whether the associated object(s) itself has associations with the `:dependent` option set. For an example of the use of the `:dependent` option, consider the following:

```
class Contact < ActiveRecord::Base
    belongs_to :user
    has_one :account_contact, :dependent => :destroy
    has_one :address, :dependent => :destroy
end
```

The `Contact` class has two relations with the `:dependent` option set. Thus, when a `Contact` object is deleted, the objects in these two relations, `account_contact` and `address`, will also be deleted. Further, since the `:dependent` option is set to `:destroy` for both these relations, any relations with the `:dependent` option set in the `AccountContact` and `Address` classes will also have their objects deleted.

2.3 Data Model Properties

The three basic relations and the four options that we have discussed above form the essence of the Rails data models. Note that using these constructs a developer can specify complex relationships among objects of a Rails application. Since a typical application would contain dozens or may be hundreds of object classes with many relationships among them, it is possible to have errors and omissions in the specification of the data model that may result in unexpected behaviors and bugs. Our goal in this paper is to develop an automated verification tool that can automatically analyze a Rails data model and identify errors. In order to look for errors though, we still need a specification of the properties that the developer expects to hold in the data model.

For example, for the FatFreeCRM running example, one property that we might want to check is the following: *"Is it possible to have an account without any contacts?"* The developer wants this behavior to be possible and may want to check that the data model is not *over-constrained* to prevent this behavior. Note that for this type of property finding an instance of the data model that satisfies the property would serve as proof of the property. However if we explore a set of instances and cannot find an instance that satisfies this property, we cannot definitely claim that the property fails

since there might be an instance that satisfies the property that we have not explored.

As another example, consider the following property: *"When an account is deleted, there will not be any contacts left without an account."* I.e., all the contacts that are related to an account must be deleted with that account. The developer may want to make sure that the data model is not *under-constrained* such that a violation of this property is possible. For this type of property, if we find two consecutive instances of the data model such that one leads to the other by an account deletion and the second one violates the stated property, then we can be sure that the property is violated and there is an error in the data model. However, if we cannot find a violation of this property in a set of instances that we analyze, then that does not constitute a proof of the property.

3. FORMALIZING DATA MODELS

Based on the Active Record files in a Rails application, we can construct a formal data model representing the objects and their relationships in that application. Note that this data model is mainly a static model. In our analysis we do not model operations that update the objects and their relationships (and corresponding database records based on the object-relational mapping) except the delete propagation that is declared 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 the rest of this section we present a formal data model and formalize the data model verification problem.

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 1 will identify the following set of object classes {Account, AccountContact, Address, Contact, User} and the relations among these object classes {account-account_contacts, account-contacts, account-address, contact-account_contact, contact-address, user-accounts, user-contacts} 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). For example, let us assume that o_A and o_{AC} are the set of objects for the Account and AccountContact classes and r_{A-AC} is an instance of the relation account-account_contact. Since this is a one-to-one (i.e., one-to-zero-or-one) relation according to the declarations in Figure 1, then we would have the following relational constraint C:

$$\begin{aligned} & \forall ac \in o_{AC},\ \exists (a, ac) \in r_{A-AC} \\ \wedge\ & \forall ac \in o_{AC}\ ((a, ac) \in r_{A-AC} \wedge (a', ac) \in r_{A-AC}) \Rightarrow a = a' \\ \wedge\ & \forall a \in o_A\ ((a, ac) \in r_{A-AC} \wedge (a, ac') \in r_{A-AC}) \Rightarrow ac = ac' \end{aligned}$$

Note that this constraint states that each Account object must be associated with zero or one AccountContact object based on this relation, and each AccountContact object must be associated with exactly one Account object. If the relation r_{A-AC} satisfies the above constraint, then we would state that $r_{A-AC} \models C$.

The dependency constraints in D 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 (based on the `:dependent` option). So, in order to determine if a dependency constraint holds, we need two instances of the same relation, say r and r', one denoting the instance before the deletion, and one denoting the instance after the deletion, respectively. Then, if the pair of relations (r, r') satisfy the dependency constraint, we write $(r, r') \models D$.

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

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

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

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

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): these are properties that we expect to hold for each instance of the data model; 2) *behavior assertions* (denoted by A_B): these are properties that we expect to hold for each pair of instances that form a behavior of the data model; 3) *state predicates* (denoted by P_S): these are predicates we expect to hold in some instance of the data model; and, finally, 4) *behavior predicates* (denoted by P_B): these are predicates we expect to hold in some pair of instances that form a behavior of the data model. We will denote that a data model satisfies an assertion or a predicate as $M \models A$ or $M \models P$, respectively. Then, we have the following formal

definitions for these four types of properties:

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

Bounded Verification of Data Models. The data model verification problem is, given one of these types of properties, determining if the data model satisfies the property. Since the number of objects in a data model is not bounded, we cannot enumerate all the instances in a data model. One possible approach is to use bounded verification where we check the property for instances within a certain bound. This is the approach we take in this paper. 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, we can check the following condition for example:

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

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 know 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:

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

and 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 [16] is a SAT-based bounded verification tool for analyzing object-oriented data models. Alloy language allows 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 above. In order to do bounded verification of Rails data models, we implemented an automated 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. Below we describe how we translate the Active Record specifications to the Alloy language.

4. TRANSLATION TO ALLOY

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 source code called ParseTree [18]. ParseTree extracts the parse tree for an entire Ruby class and returns it as an s-expression. S-expressions are generated for each model file that contains a class that inherits from ActiveRecord. We then create an s-expression processor (which inherits from SexpProcessor, the basic s-expression traversal class provided with ParseTree) to traverse the generated s-expressions and translate them to a single Alloy specification file.

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 Alloy. The inheritance relationships in Rails, such as `class Child < Parent`, are translated to Alloy using the `extends` keyword, as in `class Child extends Parent`.

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, the mapping of the Rails relationships to Alloy is as follows:

`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`
	`}`

Furthermore, one has to add a fact block that connects each pair of declarations. For the one-to-many relationship this 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` [16].

The :through Option. To translate the `:through` option, we follow the mapping from the table in the previous paragraph. 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. This is because all the fields referred to in the fact refer to the those inside that single signature. So, for the following Rails models:

```
class Account < ActiveRecord::Base
    has_many :account_contacts
    has_many :contacts, :through => :account_contacts
end
```

```
class Contact < ActiveRecord::Base
    has_one :account_contact
    has_one :account, :through => :account_contact
end
class AccountContact < ActiveRecord::Base
    belongs_to :account
    belongs_to :contact
end
```

the Alloy translation looks as follows:

```
sig Account {
    account_contacts:  set AccountContact,
    contacts:  set Contact
} { contacts = account_contacts.contact }
sig Contact {
    account_contacts:  lone AccountContact
    account:  lone Account
} { account = account_contacts.account }
sig AccountContact {
    account:  one Account,
    contact:  one Contact
}
fact {
    Account <:  account_contacts =
        ~(AccountContact <:account)
    Contact <:  account_contact =
        ~(AccountContact <:contact)
}
```

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 models:

```
class Account < ActiveRecord::Base
    has_one :address,
          :conditions => "address_type='Billing'"
end
class Address < ActiveRecord::Base
    belongs_to :account
end
```

we translate it to Alloy by abstracting the set of addresses for which the condition `address_type='Billing'` holds, to the set `Billing_Address` as follows:

```
sig Account { address:  lone Billing_Address }
sig Address { account:  one Account
sig Billing_Address in Address { }
fact {
  Account <:  address = ~(Billing_Address <:  account)
}
```

The `in` keyword in Alloy creates a subset; it is used above to create the `Billing_Address` signature. Since we are not modeling the data fields of the Rails classes, we create this arbitrary subset of `Address` without specifying exactly which elements of `Address` belong in the subset (i.e. the ones which have `'Billing'` as the `address_type`). The `address` element in `Account` can now map to just this subset of `Address`. The global `fact` block establishes this mapping by confirming the `address` and `account` 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 `Contact` needs to inherit from both `Addressable` and `Subject`. 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 `Contact` is a subset of both `Addressable` and `Subject`; 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 example from FatFreeCRM. Below are a set of Rails models with a polymorphic association. `Address` has an `addressable` association that both `Account` and `Contact` refer to:

```
class Address < ActiveRecord::Base
    belongs_to :addressable, :polymorphic => true
end
class Account < ActiveRecord::Base
    has_one :address, :as => :addressable
end
class Contact < ActiveRecord::Base
    has_one :address, :as => :addressable
end
```

The first step in translating these models is to create a common base class that all classes extend, as follows:

```
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 `Addressable` and it will contain the `has_one` relation, translated as described earlier:

```
sig Addressable extends ActiveRecord {
    address:  lone Address
}
```

Next, the relationship between `Addressable` and the target classes (`Contact` and `Account`) will be established via `facts`. Specifically, we will state that the target classes are subsets of `Addressable`, 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 `facts` 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 `facts` to state that `Addressable` is disjoint from all other non-target classes in `ActiveRecord`. The final Alloy translation is given below.

```
abstract sig ActiveRecord {}
sig Address extends ActiveRecord {
    addressable:  one Addressable
}
sig Account extends ActiveRecord {}
sig Contact extends ActiveRecord {}
sig Addressable in ActiveRecord {
    address:  lone Address
}
fact {
    Account in Addressable
    Contact in Addressable
    all x0:  Addressable | x0 in Account or
                           x0 in Contact
    no Address & Addressable
}
```

4.1 Translating the Dependency Constraints

Finally we have 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 for FatFreeCRM:

```
class Account < ActiveRecord::Base
    belongs_to :user
end
class User < ActiveRecord::Base
    has_many :accounts
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 Account {}
sig User {}
one sig PreState {
    accounts:  set Account,
    users:  set User,
    relation1:  Account set -> one User
}
one sig PostState {
    accounts':  set Account,
    users':  set User,
    relation1':  Account set -> set User
}
```

The `PreState` sig contains fields `accounts` and `users` to hold objects of each type in the system. Next, it contains a field `relation1` to hold the related `Account` and `User` objects. The product operator, `->`, produces a mapping between `Accounts` and `Users`. The multiplicity keyword `set` tells Alloy that `relation1` maps each `User` object to zero or more `Account` objects, and the keyword `one` tells Alloy that every `Account` 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 (e.g. `belongs_to :user` produces `one User` and `has_many :accounts` produces `Account set`). 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 an `Account`. To start, we define the `deleteAccount` predicate to accept a `PreState` object, a `PostState` object and an `Account` object as parameters. The body of the predicate begins by stating that `s`, the `PreState` object, contains all existing objects:

```
pred deleteAccount [s:  PreState, s':  PostState,
x:  Account] {
    all x0:  Account | x0 in s.accounts
    all x1:  User | x1 in s.users
```

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

```
    s'.accounts' = s.accounts - x
    s'.users' = s.users
    s'.relation1' = s.relation1 - (x <:  s.relation1)
}
```

Here we have deleted `x`, the `Account` object, by removing it from the set of `Account` 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 relation1'.

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_many declaration, relation1 would remain unchanged: s'.relation1' = s.relation1.) 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 andhas_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 (as we did in our case studies).

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 Account < ActiveRecord::Base
    has_many :account_contacts
    has_many :contacts, :through => :account_contacts
end
class AccountContact < ActiveRecord::Base
    belongs_to :account
    belongs_to :contact
end
class Contact < ActiveRecord::Base
    has_one :account_contact
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 Account {}
sig AccountContact {}
sig Contact {}
one sig PreState {
    accounts:  set Account,
    account_contacts:  set AccountContact,
    contacts:  set Contact,
    relation1:  Account one -> set AccountContact,
    relation2:  AccountContact lone -> one Contact,
    thru_relation = relation1.relation2
}
one sig PostState {
    accounts':  set Account,
    account_contacts':  set AccountContact,
    contacts':  set Contact,
    relation1':  Account set -> set AccountContact,
    relation2':  AccountContact set -> one Contact,
    thru_relation' = relation1.relation2
}
```

The new idea is the translation of the relation with the :through option set, the one between Account and Contact. We use the join operator, ., to define thru_relation to be the join of the other two relations.

The definition of the delete predicate (provided below) is also basically the same. Let's say we want to delete an AccountContact. We delete the object from PostState's set of account_contacts. We also delete it from any of the object's :belongs_to or :has_and_belongs_to-many relations. Incidentally, both of its relations are :belongs_to; thus, tuples containing x in both the relations relation1 and relation2 are removed. Note that thru_relation does not need to be updated explicitly; it will be updated automatically because it is defined using relation1 and relation2.

```
pred deleteAccountContact [s:  PreState, s':  PostState,
x:  AccountContact] {
    all x0:Account | x0 in s.accounts
    all x1:AccountContact | x1 in s.account_contacts
    all x2:Contact | x2 in s.contacts
    s'.accounts' = s.accounts
    s'.account_contacts' = s.account_contacts - x
    s'.contacts' = s.contacts
    s'.relation1' = s.relation1 - (s.relation1 :> x)
    s'.relation2' = s.relation2 - (x <:  s.relation2)
}
```

The :conditions, :polymorphic, and :dependent 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. A relation with the :dependent option is translated using the guidelines we discussed above; the only change is in the delete predicate. When updating the relations by removing those belonging to the deleted object, we also update relations of its associated object(s) based on the use of the :dependent option.

5. EXPERIMENTATION

We used our Active Records to Alloy translator tool and the Alloy Analyzer to analyze data models of two open source Ruby on Rails applications, TRACKS [21] and Fat Free CRM [12].

TRACKS is an application to manage things-to-do lists. It has 6062 lines of code, with 44 total classes and 13 data model classes. TRACKS allows users to organize to-do items by context or project. Notes can be added to to-do lists and projects. TRACKS is also multi-user. The Alloy specification that our translator produced for this application contains 444 lines of code.

Fat Free CRM is roughly twice as big as TRACKS, with 12069 lines of code, 54 classes and 20 data model classes. The Alloy translation has 1518 lines of code. Fat Free CRM aims to be a lightweight solution to customer relationship management (CRM). Fat Free CRM, offers the management of leads (a person who is a potential customer), accounts, opportunities and campaigns All of this is handled within a multi-user environment.

Property Classification. The model files of these two applications were fed into our automatic translator, which generates an Alloy specification. To this, we added properties about the data model and relations between objects. We had four basic classes of properties, which we describe below:

I. Relationship Cardinality: These properties check the cardinality of a relationship. For instance, the application specification may require there be a one-to-one relationship between two objects. This can be the cause of bug if the Rails programmer is not aware that Ruby on Rails' one-to-one relationship is actually a one-to-zero-or-one relationship.

II. Transitive Relations: This class of properties check whether the set of objects in a direct relationship between two objects is the same as the set of objects obtained from an indirect, or transitive, relationship. This class of properties can fail due to incorrect usage of the `:through` option. In TRACKS for instance, Notes belong to Users; Notes also belong to Projects, and Projects belong to a User. So one can check whether the User of a Note is the same as the Note's Project's User, for all Notes.

III. Deletion Does Not Create Orphans: We also verified properties that checked whether deleting an object caused a related object to be orphaned. This happens when semantically two objects should always be related, but the application allows one of them to be deleted.

IV. Deletion Does Not Cause Dangling References: Another class of properties we checked were whether deleting an object caused dangling references. This may happen due to forgotten `:dependent` declarations on the `has_many` or `has_one` side of a relationship.

V. Deletion Propagates to Associated Objects: Finally, we checked whether deleting an object also caused certain related objects to be deleted (or that the associated objects did not get deleted if they should not have). Again, this type of properties may fail due to incorrect usage of the `:dependent` option. One can also verify that an associated object still exists after the delete.

The complete list of properties verified for both applications is shown in Table 1. It includes a description of the property, whether it passed or failed during verification, and which of the above categories it belongs to. It also includes the type of property based on the types we defined in Section 3.

Property Classification. In the TRACKS Application, a total of ten properties were checked. Of the ten properties, five failed. Of these that failed we considered it a data modeling error if the property could have been enforced by the data model but was not.

The first property that failed is property T4. Inspection of the data models reveals that the relationship between a Todo and Projects is one-to-many; the `has_many/belongs_to` declaration pair is used. However, the application actually requires a zero-or-one-to-many relationship between Todo and Projects. Looking at the data models shows that to make up for Rails' limited expressive power for relations, the application programmer has enforced this property in the data model by adding code to return a NullProject.[2] This code is not part of the static data model so our analysis does not model it. Since this property cannot be enforced in a Rails data model we do not consider this failure a data-modeling error.

[2]An empty Project that represents the absence of an object. This is preferable over a null reference in some cases.

Type	**TRACKS Properties**	
I,A_S	T1 Every Todo has a Context	P
I,P_S	T2 A Context may have no Todos	P
I,A_S	T3 A Todo must have a Context	P
I,P_S	T4 A Todo can have no Project	F
II,A_S	T5 Note's User = Note's Project's User	F
I,A_S	T6 Every User has a Preference	F
IV,A_B	T7 No dangling Todos after User delete	P
III,A_B	T8 No orphan User after Preference delete	F
IV,A_B	T9 No dangling Todos after Context delete	P
IV,A_B	T10 No dangling Todos after User delete	F
Type	**FFCRM Properties**	
I,A_S	F1 An Opportunity must have a Campaign	P
I,A_S	F2 A Task must have a User	P
I,P_S	F3 An Account may have no Activities	P
I,A_S	F4 At most one User per Lead	P
II,A_S	F5 A ContactOpportunity's Opportunity = ContactOpportunity's Contact's Opportunity	P
I,P_S	F6 A Contact may have no Tasks	P
I,A_S	F7 A Contact cannot have two Accounts	P
II,A_S	F8 User's Opportunities = User's Campaigns' Opportunities	F
II,A_S	F9 Lead's User = Lead's Activities' User	P
IV,A_B	F10 No dangling Contacts after Account delete	P
V,A_B	F11. Deleting an Account does not delete its Contacts	P
V,A_B	F12 Deleting an Account deletes its Todos	P
V,A_B	F13 Deleting a Campaign deletes its Leads	P
V,A_B	F14 Deleting a Campaign deletes its Opportunities	P
V,A_B	F15 Deleting Campaign deletes its Leads' Tasks	P
IV,A_B	F16 No dangling Contacts after Lead delete	F
V,A_B	F17 Deleting a Lead does not delete its Contacts	P
IV,A_B	F18 No dangling Users after Lead delete	P
V,A_B	F19 Deleting a Lead does not delete its User	P
V,A_B	F20 Deleting a User means its associated Activities are also deleted	P

Table 1: Verification Results (P:Pass, F: Fail)

The next property to fail is T5. In the TRACKS application, there is a relationship between Users and Projects, Users and Notes, and Notes and Projects. This property checks that the User a Note belongs to is the same User that the Note's Project is associated with. Analysis of the failure of this property shows that there is a flaw in the design of the data model. The `user` field is duplicated in the Project and Note classes. This was probably done for ease of access of the User from the Note class, as opposed to going through the Project class. Currently, this property is being upheld in the application by the code placed in the controller. However, in the future if the controller code is changed and the programmer is unaware of this invariant, a bug may be introduced into the application. This property could have been enforced in the data model using the `:through` option, hence we consider this failure a data-modeling error.

The third property that failed verification in TRACKS is property T6. In the data models, one observes that the relationship between User and Preference is set up using `has_one` and `belongs_to`. Recall that this is a one-to-zero-or-

one relationship. However, in TRACKS, this relationship is actually one-to-one. Once again we observe the limitation of Rails' ability to express relationships of this kind. Although this constraint is being enforced in the controller, there are ways of enforcing the one-to-one relationship in the data models. For example, the User and Preferences tables can be merged or a construct in Rails called callbacks may be used. Consequently, this property failure also points out a data-modeling error in the application. Property T8 failed due to this same data modeling error. A User should always be related to a Preference, but allowing a Preference to be deleted breaks this property.

The final property that failed verification in TRACKS is property T10. In the data models, we see that there is a relationship where a User `has_many` Contexts, and a Context `has_many` Todos. Because the `has_many` relation in User has the `:dependent` option set to `:delete_all`, the associated Context objects will be deleted, but Rails will not go into Context and look at its relations. Thus, the User deletion does not get propagated into the relations of the Context object, including the Todos that this property is checking for. Hence, this property pinpoints another data-modeling error. However, the User object itself also is related to Todos, and since the application has controller logic that enforces this set of Todos is the same as the ones obtained by navigating through the Context relation, the application currently does not manifest this data modelling error.

For the Fat Free CRM application, twenty properties were verified. Of these, two did not pass the verification. The first is F8, and the situation for this property is the same as property T5 of TRACKS. Hence this property points to a data-modeling error.

The other property that fails for Fat Free CRM is property F16. Leads and Contacts are associated such that a Lead `has_one` Contact, with no `:dependent` option set on the association. Thus, when a Lead is deleted, the associated Contact still exists. A comment next to the relation declaration confirms that the application programmer wanted the Contact to remain alive. However, this Contact now has a null reference and thus breaks the model design; the use of the `has_one/belongs_to` declaration pair signifies a one-to-zero-or-one relationship, whereas it seems as if the application programmer wanted a zero-or-one-to-zero-or-one relationship. Further inspection of the code reveals that this relationship serves no purpose. In the application logic, some Leads may be upgraded to Contacts, which means (in the code) that a Contact object is created from a Lead object by copying many of the fields over. Hence, a Contact never refers back to the Lead that created it, nor the Lead to the Contact it created. Perhaps this relationship seemed necessary during the initial drafts of the data model design; however, it is now extraneous and only allows room for future bugs. Thus, the violation of this property uncovers a data-modeling error.

Performance. To measure performance, we recorded the amount of time it took for Alloy to run and check the properties on a bounded number of instances, as well as the number of variables and clauses generated in the boolean formula generated for the SAT-solver. The time, number of variables and number of clauses are averaged over the properties for each application. These performance measures are taken over an increasing bound, from at most 10 objects for each class to at most 35 objects for each class. This means that each property was checked by instantiating up to 10 instances of each object, up to 15 of each object, etc The results are summarized in Figure 2. We see that the number of variables and clauses, and verification time increase with increasing bound as expected. In the worst case, this increase could be exponential. However, the slowest verification time (for a bound of 35 objects) being only 22 seconds confirms that the bounded analysis we are performing is feasible for analyzing properties of real-world web applications.

6. RELATED WORK

Formal modeling and automated verification of web applications has been investigated before. There has been some work on analyzing navigation behavior in web applications, focusing on correct handling of the control flow given the unique characteristics of web applications, such as the use of a browser's "back" button combined with the stateless nature of the underlying HTTP protocol [17]. 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 [20] and investigated further later on [14, 2, 13]. State machine based models have been used to automatically generate test sequences [23], perform some form of model checking [19] and for runtime enforcement [13]. In contrast to these previous efforts, we are focusing on analysis of the data model rather than the navigational aspects.

There has been some prior work on formal modeling of web applications using UML [7] and extending UML to capture complex web application behavior such as browsing and operations on navigation states [1]. WebML [5] is a modeling language developed specifically for modeling web applications. 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 [6]. 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.

The verification of traditional, non-MVC web applications, has also been investigated in recent years [15, 9, 10, 11]. In this paper, by focusing on MVC style web applications we are able to exploit the modularity in the MVC architecture and extract formal data models from existing applications.

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 [8]. 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 [22]. However, unlike our work, the translation to Alloy is not automated in these earlier efforts. Finally, Alloy has also been used for discovering bugs in web applications related to browser and business logic interactions [3]. This is a different class of bugs than the data model related bugs we focus on in this paper.

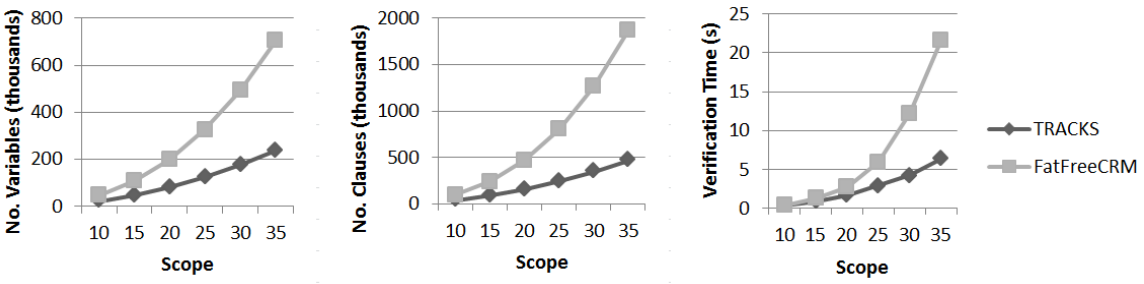

Figure 2: Verification Performance with Increasing Bound (Scope)

7. CONCLUSIONS

We presented techniques for bounded verification of Ruby on Rails data models. We showed that by exploiting the inherent modularity in MVC frameworks it is possible to extract a static data model from a Rails application. We formalized the bounded verification problem for Rails data models and realized a bounded verification framework by implementing an automated translator from Rails data models to Alloy language. The Alloy specification that our translator outputs can then be appended with properties about the data relationships and checked by the Alloy Analyzer. We applied this approach to two open source web applications and demonstrated that this type of bounded verification is feasible and can discover data modeling errors in real applications. As future work we plan to investigate fusing the data model investigated in this paper with a navigation model in order to analyze dynamic data model behavior.

8. REFERENCES

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

[2] 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.

[3] 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.

[4] F. Buschmann et al. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.