Inductive Verification of Data Model Invariants for Web Applications *

Ivan Bocić and Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara, USA
{bo,bultan}@cs.ucsb.edu

ABSTRACT

Modern software applications store their data in remote cloud servers. Users interact with these applications using web browsers or thin clients running on mobile devices. A key issue in dependability of these applications is the correctness of the actions that update the data store, which are triggered by user requests. In this paper, we present techniques for automatically checking if the actions of an application preserve the data model invariants. Our approach first automatically extracts a data model specification, which we call an *abstract data store*, from a given application using *instrumented execution*. The abstract data store identifies the sets of objects and relations (associations) used by the application, and the actions that update the data store by deleting or creating objects or by changing the relations among the objects. We show that checking invariants of an abstract data store corresponds to *inductive invariant verification*, and can be done using a mapping to *First Order Logic (FOL)* and using a FOL theorem prover. We implemented this approach for the Rails framework and applied it to three open source applications. We found four previously unknown bugs and reported them to the developers, who confirmed and immediately fixed two of them.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Design, Verification

Keywords

Automated verification, data model, inductive invariants, Ruby on Rails

*This work is supported by the NSF grant CCF 1117708.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '14, May 31 - June 7, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2756-5/14/05 ...$15.00.

1. INTRODUCTION

Nowadays, most computers are connected to the Internet. This network connectivity, when combined with the increasingly prevalent cloud computing platforms, enables software applications to store data on remote servers and use thin clients (web browsers or mobile applications) that provide access to application data from any device, anywhere, anytime, without maintaining any local copies. However, these applications are challenging to develop and maintain since they are complex software systems consisting of distributed components that run concurrently and interact over the Internet.

In order to reduce this complexity and achieve modularity, most web application development frameworks use the Model-View-Controller (MVC) pattern [25] to separate the code for the data model (Model) from the user interface logic (View) and the navigation logic (Controller). The key component in applications that use this pattern is the data model, which is responsible for identifying the sets of objects and the relations (associations) among the objects stored in the back-end data store. Correctness of *actions* that update the data store is the most significant correctness concern for these applications since erroneous actions can lead to unrecoverable corruption or loss of data.

Another widely used approach in web application development is the Representational State Transfer (REST) architecture with RESTful interfaces. In RESTful applications, any action on the data model can be invoked at any time and any number of times. This implies that any property that should hold on the data model must be preserved by each action, independently of the history of previous action executions. Hence, invariants of the data model can be checked in a modular manner, by checking each action in isolation, using inductive invariant verification.

Based on the above observations, we developed a novel approach for automatically verifying data model invariants of web applications. First, by exploiting the structure of the MVC-pattern, we automatically extract an abstract specification of the data model, including actions that update the data store, using instrumented execution. Next, we convert verification queries about the data model (stated as invariants) to formulae in First-Order-Logic (FOL) based on inductive invariant verification. Finally, we use an automated FOL theorem prover to determine the result of the verification query.

We implemented our approach (Figure 1) for the Rails framework [32]. We decided to focus on the Rails framework since it is widely used. However, the approach presented in

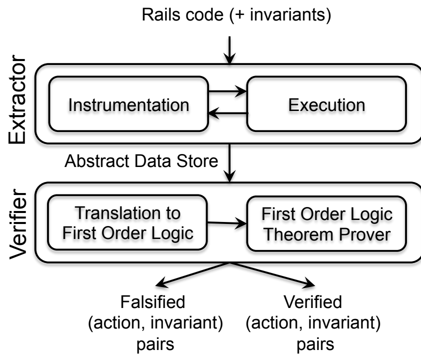


Figure 1: Verification of data model invariants

this paper can be adapted to other MVC-based web application frameworks such as Django [9] and Spring [35]. We evaluated our approach on three open source Rails applications: Kandan [23], Tracks [37], and Fat Free CRM [10]. These are non-trivial applications with more than 2K, 18K, and 30K lines of code respectively. In addition to proving that the majority of the actions preserve the stated invariants, our tool also discovered previously unknown bugs in these applications.

Our contributions in this paper are: 1) An intermediate representation for data model specifications called abstract data stores. 2) Data model extraction using instrumented execution and its implementation for the Rails framework. 3) Reduction of invariant verification queries for abstract data stores to FOL using inductive invariant verification, and a translator that implements this reduction. 4) Implementation of a verification tool for the Rails framework that combines the components listed above with an existing FOL theorem prover. 5) Evaluation of the proposed approach on real world applications, and experiments demonstrating the performance and effectiveness of our verification tool for these applications.

This paper is organized as follows. Section 2 provides an overview of the problem we are addressing and the techniques we use to solve it. Section 3 gives an overview of our intermediate language (abstract data stores) for representing data model specifications. Section 4 explains our instrumented execution technique for extracting abstract data store specifications from application code. Section 5 explains how we translate invariant verification queries about abstract data store specifications into first order logic. Section 6 presents our experimental results and the bugs we found. Section 7 includes discussions about the related work and Section 8 concludes the paper.

2. OVERVIEW

Figure 2 shows an excerpt from an example Rails application. In Rails, the data model is implemented using an Object-Relational Mapping (ORM) library called ActiveRecord. It is very similar to other ORM libraries (Hibernate, Django ORM etc.) in that each class corresponds to a table in a relational database, and that object references map to foreign keys between tables. What sets ActiveRecord apart is the highly dynamic implementation of the library where methods get generated dynamically from the schema. Some methods are generated on invocation from the method name (e.x. `User.find_by_name` will, at invocation, generate

```
1  class User
2    has_many :users
3    has_many :projects
4  end
5  class Project
6    belongs_to :user
7    has_many :todos
8    has_many :notes
9  end
10 class Todo
11   belongs_to :user
12   belongs_to :project
13 end
14 class Note
15   belongs_to :project
16 end
17 class TodosController
18   def create
19     @project = Project.find(params[:project_id])
20     @user = User.find(params[:user_id])
21     @todo = Todo.new
22     @todo.user = @user
23     @todo.project = @project
24     @todo.save!
25     respond_to(...)
26   end
27 end
28 class ProjectsController
29   def destroy
30     @project = Project.find(params[:project_id])
31     @project.notes.each do |n|
32       n.delete
33     end
34     @project.delete
35     respond_to(...)
36   end
37 end
```

Figure 2: Excerpts from a Rails application

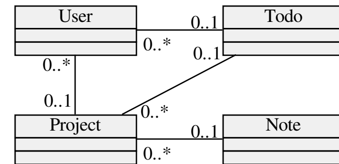


Figure 3: Class diagram for the example

a method that looks up a `User` by name). Some methods, called *ghost methods*, may not exist even after invocation.

The example application defines four ActiveRecord classes: `User`, `Project`, `Todo` and `Note` are declared in lines 1-4, 5-9, 10-13 and 14-16 respectively. Each class contains a set of relations (associations) with other classes. These associations are declared using methods `belongs_to`, `has_one`, `has_many` and `has_and_belongs_to_many` that differ in cardinality and schema details. Figure 3 shows the class diagram corresponding to the code given in Figure 2. For example, each `Todo` object has at most one associated `Project` (line 12). The types and symmetry of associations are inferred from association names; for example, for every `Project` p and every `Todo` t of that `Project`, the `Project` of t is p.

This application contains two actions: one in `TodosController` called `create` and one in `ProjectsController` called `destroy`. The `ProjectsController#create` action (lines 18-26) takes two arguments as part of the request called `project_id` and `users_id`. These arguments are used to lookup the corresponding `User`/`Project` objects and assign them to variables (lines 19-20). The action, then, creates a new `Todo` instance (line 21), associates it with the queried objects (lines 22 and 23) and saves the changes (line 24). The response is synthesized in line 25 by the view (which is omitted for brevity).

The `TodosController#destroy` action (lines 29-36) takes a single request argument `project_id`. After looking up the

`Project` object under that id (line 30), it iterates through all `Notes` associated with that project (line 31) and deletes them (line 32). Finally, said project gets deleted (line 34).

Our Verification Approach.

Assume that we would like to verify the following property for the application in Figure 2: *Each* `Todo` *object is associated with a* `Project` *object.* What we mean by verifying this property is that, we would like to show that this property holds for all configurations of the data store during all possible executions of this application. For a Rails application, this means verifying that all actions that update the data store preserve the given property.

In order to do that, we first need a formal specification of the property. We developed a Rails library for specification of data model invariants using Rails syntax. For example, the above property would be stated as:

```
invariant forall{ |todo| not todo.project.empty? }
```

Given the property specification, our verification approach consists of two major phases (Figure 1). The first phase is automatic extraction of a formal specification, which we call an *Abstract Data Store (ADS)*, that characterizes the model and the actions of the input web application. ADS models the data store as sets of objects (corresponding to objects of the data model classes) and relations among them (corresponding to the associations among the data model classes). Attributes that correspond to basic types are not modeled (i.e., they are abstracted away). This means that we can verify invariants about sets of objects and relations among them (like the example above), but not about numeric attributes of objects for example.

The crucial part of the extraction phase is extraction of action specifications, where actions of the web applications are translated to ADS actions. ADS actions contain constructs for creating and deleting objects and updating relations, and they allow non-determinism, which is necessary due to abstraction of the attributes with basic types. For example, `if` statements are represented as non-deterministic choices.

We exploit the MVC-pattern during the extraction phase. Actions that update the data model correspond to Controller actions that are executed in response to user requests and can in turn execute methods of the Model. We ignore the View construction since it does not influence the data store state.

Statically extracting action specifications from Rails code is challenging due to the dynamic nature of the Ruby language. To address this challenge we automatically instrument the application code and extract the model by executing the instrumented code.

The second major phase of our approach is automated verification of invariants on the extracted model using First Order Logic (FOL). Since Rails applications are RESTful they allow any action to be invoked at any time. This means that all actions have to preserve all invariants of the data model. We therefore do inductive verification of invariants. For each action/invariant pair we generate a FOL theorem (i.e., a formula with no free variables) that evaluates to true if and only if the invariant is preserved by the action. These FOL theorems are sent to a theorem prover to check if they evaluate to true (which means that the invariant holds) or false (which means that the invariant is violated).

For the example application shown in Figure 2, and the invariant we mentioned earlier, we generate two formulae. The theorem prover shows that the formula for the `TodosController#create` action evaluates to true (which means that the action preserves the given invariant) whereas the formula generated for the `ProjectsController#destroy` action evaluates to false (which means that this action violates the invariant).

3. ABSTRACT DATA STORES

Abstract Data Store (ADS) is an intermediate language we use for representing the data models we extract from web applications. We first start with a brief overview of the semantics of ADS specifications and then we describe the constructs of the ADS language.

3.1 Formal Model

Semantically, an abstract data store is a structure $DS = \langle C, R, A, I\rangle$ where C is a set of classes, R is a set of relations, A is a set of actions, and I is a set of invariants.

The set of classes C identifies the types of objects that can be stored in the data store. Each class can have a single superclass or no superclass (superclass$(c) \in C \cup \{\bot\}$) and, transitively, the superclass relation cannot contain cycles. A relation $r = \langle c_o, c_t, card\rangle \in R$ contains an origin class $c_o \in C$, a target class $c_t \in C$ and a cardinality constraint $card$ (such as one-to-one, one-to-many etc.).

3.1.1 Data Store States

Given a data store $DS = \langle C, R, A, I\rangle$, the set of all possible *data store states* is denoted as $\overline{DS}$. Each data store state is a structure $\langle O, T\rangle \in \overline{DS}$ where O is a set of *objects* and T is a set of *tuples*.

Objects are instances of classes, whereas tuples are instances of relations. Each object $o \in O$ is an instance of a class $c \in C$ denoted by $c = \text{classof}(o)$. Each tuple $t \in T$ is in the form $t = \langle r, o_o, o_t\rangle$ where $r = \langle c_o, c_t, card\rangle \in R$ and $\text{classof}(o_o) = c_o$ and $\text{classof}(o_t) = c_t$. For a tuple $t = \langle r, o_o, o_t\rangle$ we refer to o_o as the origin object and o_t as the target object. Cardinality constraints of each relation $r \in R$ must be satisfied by every data store state in $\overline{DS}$.

3.1.2 Actions and Invariants

Given a data store $DS = \langle C, R, A, I\rangle$, A denotes the set of actions. Each action $a \in A$ corresponds to a set of possible state transitions $(\langle O, T\rangle, \langle O', T'\rangle) \subseteq \overline{DS} \times \overline{DS}$. Actions characterize updates to the data store states such as creation or deletion of a set of objects or creation or deletion of a set of tuples.

Given a data store $DS = \langle C, R, A, I\rangle$, I is the set of invariants. An invariant $i \in I$ corresponds to a Boolean function $i\colon \overline{DS} \to \{\text{false}, \text{true}\}$ that identifies the set of data store states which satisfy the invariant.

3.1.3 Behaviors

Given a data store $DS = \langle C, R, A, I\rangle$, a behavior of a DS is an infinite sequence of data store states $\langle O_0, T_0\rangle$, $\langle O_1, T_1\rangle$, $\langle O_2, T_2\rangle, \ldots$ where

- for all $k \geq 0$, $\langle O_k, T_k\rangle \in \overline{DS}$ and there exists an action $a \in A$ such that $(\langle O_k, T_k\rangle, \langle O_{k+1}, T_{k+1}\rangle) \in a$, and
- $\forall i \in I : i(\langle O_0, T_0\rangle) = \text{true}$

In other words, each behavior of a data store starts with an initial data store state where all invariants hold, and each pair of consecutive states corresponds to execution of a data store action.

Category	Node	Children	Semantics
Statement	`Block`	**Statement*	Executes statements sequentially
	`Either`	*`Block`	Executes exactly one of the children `Blocks`
	`ObjectSetStmt`	*Object Set*	Evaluates the object set. Used for object sets with side-effects (e.g. `CreateObjectSet`)
	`Assign`	`Variable`, *Object Set*	Assigns the object set to the `Variable`
	`Delete`	*Object Set*	Deletes objects belonging to the object set; disassociates deleted objects from all other objects
	`CreateTuple`	*Object Set*, `Relation`, *Object Set*	Associates all objects from the two object sets over the `Relation`
	`DeleteTuple`	*Object Set*, `Relation`, *Object Set*	Disassociates all objects from the two object sets over the `Relation`
	`ForEach`	`Variable`, *Object Set*, `Block`	Executes the `Block` once for each object in the given object set, assigning the singleton set of this object to the `Variable` prior to each iteration
Object Set	`Variable`		Contains objects assigned to the `Variable` and were not deleted since
	`CreateObjectSet`	`Class`	Creates a new object of the stated `Class` that is not associated to any other object, and has the value of the singleton set containing this object
	`AllOfClass`	`Class`	Contains all objects of class `Class`, including subclasses
	`Subset`	*Object Set*	Contains a subset of the given object set
	`OneOf`	*Object Set*	Exactly one object from the given object set, or empty if the given object set is empty
	`Union`	**Object Set*	Union of the given object sets
	`Empty`		Contains no objects. The type of this object set is compatible with all other types in `Union` etc
	`Dereference`	*Object Set*, `Relation`	All objects associated with at least one object from the given object set over the given `Relation`
	`DereferenceCreate`	*Object Set*, `Relation`	Creates a new object and associates it with all objects from the supplied object set over the given `Relation`. Returns the newly created object

Figure 4: Abstract Data Store Statement and Object Set Nodes

Given a data store $DS = \langle C, R, A, I \rangle$, all states that appear in a behavior of DS are called the *reachable states* of DS and denoted as $\overline{DS}_R$.

3.2 Language Constructs

We represent ADS specifications as abstract syntax trees (since ADS is an intermediate representation this is sufficient). The abstract syntax tree for an ADS specification contains a set of `Class`, `Relation`, `Action`, and `Invariant` nodes corresponding to the semantic model $DS = \langle C, R, A, I \rangle$.

Following the formal definition of C and R given in Section 3.1, a `Class` node may refer to another `Class` node as its superclass, and contains any number of `Relations`. `Relation` nodes are defined by name, target `Class` and cardinality.

The most complex part of an ADS specification are the action specifications. The `Action` nodes contain a `Block` node, which in turn contains any number of *statement* nodes. Most statements use *object sets* as arguments. Object sets serve a purpose similar to expressions in common programming languages. The list of statement and object set nodes in ADS language is given in Figure 4. The ADS language includes constructs for creating and deleting objects (`CreateObjectSet`, `DereferenceCreate`, `Delete`), updating relations (`CreateTuple`, `DeleteTuple`), variables and assignments (`Variable`, `Assign`), loops (`ForEach`), and non-determinism (`Either`).

The `Invariant` nodes include standard FOL operators including quantification over object sets.

4. MODEL EXTRACTION

The goal of the extraction phase is to translate the Rails code to an intermediate language that captures the data model behavior (in our case, the target language is the ADS language defined above). Ideally, one would do this by traversing the source code while generating the code in the target language. However, due to the dynamic nature of Ruby, we do not have all of Rails source code available statically as some methods are generated at runtime or may not exist even after being invoked and, moreover, we do not have the type information that would be necessary for the translation.

To address this challenge, we developed a technique for model extraction via instrumented execution. We do code generation at instrumented runtime, with the entire Rails stack running as configured by the application. In other words, we instrument the Rails application to implement the code generation, and run the instrumented application.

Note that, for this approach to work, we need to execute all parts of the source code during instrumented execution, which we achieve by executing both branches one after another when we encounter a branch condition. This also means that the code generation must be flow-insensitive (so that the order of execution of different branches does not influence the translation). However, this is not a problem, since order of traversal during code generation does not matter as long as the generated code follows the control flow of the source code.

We extract the data model specification by focusing on one action at a time. We initiate instrumented execution by instrumenting the action itself in such a way that, immediately before invoking any method originating from the action, the called method gets instrumented as well. This way instrumentation propagates through the program, including dynamic behavior. In effect, instrumentation and instrumented execution are interleaved.

4.1 Action Extraction Example

Below, we illustrate our approach using the example in Figure 5 where Figure 5(b) shows the instrumented version of the action shown in Figure 5(a). This action creates a new object of class `User` and assigns it to a variable `@user` (line 2a), iterates through all existing `Projects` using `project` as the iterator variable (line 3a) and, if the `project` is `hiring?` (line 4a), adds the `project` to the collection of `@user`'s projects (line 5a). The `@user` object is saved in line 8a and the response is synthesized in line 9a, with details omitted for brevity.

Before the instrumentation phase, we overload most ActiveRecord methods to return *meta-objects*. Meta-objects do not contain any concrete values from the database, but mimic actual ActiveRecord objects by responding to the same method signatures, and return other meta-objects on method calls.

These meta-objects can be translated into *object sets* defined in Figure 4. For example, calling `User.build(...)` returns a meta-object that will be translated into a `CreateObjectSet(User)` object set. Calling any `User.find_by_...(...)` method resolves to a meta-object that will be translated into a `OneOf(AllOf(User))` object set.

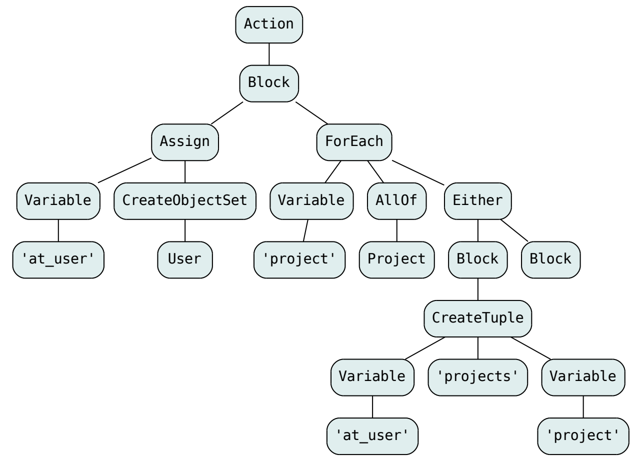
Figure 5(b) contains the instrumented version of the original action. All statements are executed within a `ins_stmt`

```
1a  def create_user_example
2a    @user = User.build(params)
3a    Project.all.each do |project|
4a      if project.hiring?
5a        @user.projects << project
6a      end
7a    end
8a    @user.save!
9a    respond_to(...)
10a end
```

(a) Example action

```
1b  def create_user_example
2b    ins_stmt(ins_asgn(:@user, User.build(params)))
3b    ins_stmt(Project.all.each do |project|
4b      ins_if(
5b        project.hiring?,
6b        ins_stmt(@user.projects << project),
7b        nil
8b      )
9b    end)
10b   ins_stmt(@user.save!)
11b end
```

(b) Action after instrumentation



(c) Extracted action specification

Figure 5: Abstract Data Store extraction example

call, which appends the ADS specification extracted from the statement to the current ADS `Block`.

Executing `ins_asgn` (line 2b) creates an `Assign` ADS node that contains the variable name and the ADS expression for the right hand side of the assignment.

Lines 3b-9b all belong to the same `ins_stmt` call. The expression `Project.all` returns a meta-object that will be translated to the `AllOf(Project)` object set. The `each` method in Ruby is normally used to iterate through a collection and execute a block for each element. When called on an meta-object, however, the block will be executed only once; the ADS `Block` node extracted from this single block execution becomes the `Block` of the extracted `ForEach` statement.

We replace every Ruby branch statement (lines 4a-6a) with an `ins_if` statement (lines 4b-8b). During instrumented execution, `ins_if` will execute both the *then* and *else* branches in different contexts and return an `Either` ADS node.

Finally, the statement in line 5a is identical to line 6b of the instrumented action. During instrumented execution all variable meta-objects will be translated into `Variable` nodes, so the association meta-object will be translated into `Dereference(Variable('@user'), 'projects')`. The `<<` method creates an `CreateTuple` statement node. The `save!` method (line 10b) is treated like a no-op. This translation results in the ADS action specification shown in Figure 5(c).

4.2 Limitations and Assumptions

Our extraction method has some limitations and works under certain assumptions. Since our extraction method is dynamic, if an application were to construct a method's code based on the user input, then we would not be able to capture the full behavior of the application. However, this is not the case for Rails actions since they are implemented with ActiveRecord which is bound by the database schema, and the dynamically generated methods follow this static schema. So, although our extraction method would not work for all dynamically generated methods in Ruby, it is able to capture dynamically generated methods in Rails actions.

We assume that the application will never generate associations using mass assignments with foreign keys. In addition, the `destroy` operation can be translated to our intermediate representation only when no circular destroy dependencies exist. This limitation comes from the fact that, in the presence of circular destroy dependencies, the delete operation propagation becomes recursive and this cannot be translated into first order logic.

Our extraction tool ignores polymorphic associations since they are not currently modeled in our intermediate representation. However, with some extra programming effort, it is possible to extend our approach to handle polymorphic associations. Similarly, the extraction assumes that all modifications to objects in an action are executed atomically, while in actuality, some changes are kept in memory until a `save` method is called. Again, it is possible to extend our approach to handle these cases.

Our current implementation does not instrument and extract `while` loops at this point, but with extra effort this can be handled in a similar way we handle the `each` loops.

We assume that there are no recursive methods used during action execution. However, our approach can be extended to handle recursive functions if our intermediate language was extended to handle function call semantics.

Since branch conditions are not guiding the branch choice during instrumented execution, it may happen that the execution takes impossible paths. For example, in Tracks, the login procedure checks whether an external authentication library is included and, if it is, delegates authentication to it. During instrumented execution this branch gets executed even if the external module is not included, causing an exception. Similar problems may arise if the branch condition checks whether a variable has a non-null value, and the branch gets executed even if this variable is null. We anticipate cases like this and drop execution paths that raise an exception during instrumented execution.

Our path exploration is not guided by inputs since we explore all paths regardless of input values. Since action inputs are strings (usually URL parameters) and are abstracted away, input information is unnecessary for the purpose of ADS extraction.

5. VERIFICATION VIA FOL

Given an abstract data store $DS = \langle C, R, A, I \rangle$, we call *DS consistent* if and only if all reachable states of DS satisfy

all the invariants of DS, i.e., DS is consistent if and only if for all $\langle O,T\rangle \in \overline{DS}_R$, for all $i \in I$, $i(\langle O,T\rangle) =$ true. The verification problem to determine if a given abstract data store is consistent. Since we do not bound the sizes of the classes and relations in a data model, and since we allow arbitrary quantification in invariant properties, determining if a data store specified in the ADS language is consistent or not is an undecidable verification problem.

As we discussed earlier, in RESTful applications, each action is required to preserve the invariants of the data model independently of the previous execution history. This is a stronger requirement that implies the consistency condition defined above, and can be formulated as inductive invariant verification. An inductive invariant is a property where given a state that satisfies the property, all the next states of that state also satisfy the property. In other words, an inductive invariant is a property that is preserved by all transitions (i.e., all actions) of a given system. An abstract data store $DS = \langle C, R, A, I\rangle$ is consistent if the conjunction of all the invariants $i \in I$ is an inductive invariant. In other words, an abstract data store $DS = \langle C, R, A, I\rangle$ is consistent if the formula, defined below, is valid:

$$\begin{aligned} F_{cons} \equiv & \forall \langle O,T\rangle \in \overline{DS}, \forall a \in A, \\ & (\forall i \in I, i(\langle O,T\rangle)) \Rightarrow (\forall i \in I, i(a(\langle O,T\rangle))) \end{aligned}$$

In the rest of this section we show that, given an ADS specification, the above formula can be checked by first translating it to First Order Logic (FOL), and then, using an automated FOL theorem prover for checking its validity. An automated FOL theorem prover can either prove that a given FOL formula is valid (corresponding to the case that the data store is consistent), or it can prove that the formula is not valid (corresponding to the case that there exists and action that violates an invariant), or the theorem prover may never terminate (due to undecidability of FOL). We handle non-termination by setting a timeout value, and when the timeout is reached we report that the result of the verification is inconclusive. In our implementation, we targeted a FOL theorem prover called Spass [38], however, the translation discussed below would work for any FOL theorem prover.

5.1 An Overview of FOL with Equality

First Order Logic (FOL) is a formal system that operates on an infinite domain of entities. A theory in FOL with equality is composed of a finite number of *predicates* and *axioms*. FOL formulae are composed of variables (that range over the infinite set of domain elements), predicate symbols (for a k-ary predicate p and variables $v_1, v_2, \dots v_k$, $p(v_1, v_2, \dots v_k)$ is a formula), Boolean operators ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) and constants (false, true), quantification (for a formula f with free variables $v_1, v_2, \dots v_k$, both $\forall v_1, v_2, \dots, v_k\colon f$ and $\exists v_1, v_2, \dots, v_k\colon f$ are formulae), and the equality predicate (for variables v_1 and v_2, $v_1 = v_2$ and $v_1 \neq v_2$ are formulae).

Axioms are formulae that set the basis of the theory: they are assumed to be true by default. A set of *conjecture* formulae holds on a FOL theory if and only if, for all predicate assignments such that all axioms are true, all conjectures are true as well.

The equality predicate has its expected interpretation (it is reflexive, symmetric and transitive, and for any two variables x_1 and x_2 such that $x_1 = x_2$, the value of any predicate does not change if any argument x_1 is replaced with x_2). Any variable that appears in a formula without being quantified is called a *free variable*. Formulae with free variables cannot be assigned a truth value or be used as axioms or conjectures.

```
class Parent1 {}
class Child1 extends Parent1 {}
class GrandChild extends Child1 {}
class Child2 extends Parent1 {}
class Parent2 {}
```

$$\forall o\colon \bigwedge \begin{pmatrix} \texttt{Child1}(o) \Rightarrow \texttt{Parent1}(o) \\ \texttt{GrandChild}(o) \Rightarrow \texttt{Child1}(o) \\ \texttt{Child2}(o) \Rightarrow \texttt{Parent1}(o) \end{pmatrix} \quad (1)$$

$$\forall o\colon \bigwedge \begin{pmatrix} \texttt{Parent1}(o) \Rightarrow \neg\texttt{Parent2}(o) \\ \texttt{Parent2}(o) \Rightarrow \neg\texttt{Parent1}(o) \\ \texttt{Child1}(o) \Rightarrow \neg\texttt{Child2}(o) \\ \texttt{Child2}(o) \Rightarrow \neg\texttt{Child1}(o) \end{pmatrix} \quad (2)$$

$$\forall o\colon \text{is_object}(o) \Leftrightarrow \begin{pmatrix} \texttt{Parent1}(o) \vee \texttt{Child1}(o) \vee \texttt{GrandChild}(o) \\ \vee \texttt{Child2}(o) \vee \texttt{Parent2}(o) \end{pmatrix} \quad (3)$$

Figure 6: Example class model and formulae

During the translation process we describe below we often conjoin or disjoin a set of formulae into a single formula. If this set happens to be empty, the resulting conjunction/disjunction is assumed to take the form of the neutral element of conjunction (true) or disjunction (false), respectively.

5.2 Translation of Classes and Relations

Classes and relations constitute the basis of an abstract data store, so we start the description of our translation with them.

Translation of Classes.

We show a simple class hierarchy in Figure 6 which is designed to concisely showcase all features of the translation for classes as well as corresponding formulae.

We introduce a unary predicate $is_object(o)$ that evaluates to true if and only if the domain element o represents an object. In addition, for each class $c \in C$, we declare a new unary predicate $c(o)$ that evaluates to true if and only if o is an instance of class c.

To enforce inheritance, for each two classes c_p and c_c such that $c_p = \text{superclass}(c_c)$ we define that all objects of the c_c type are, by implication, of c_p type as well (formula 1 in Figure 6). We allow instances of a parent class to not be of any child class. In addition, different classes with the same superclass have mutually exclusive object sets (formula 2). Finally, all objects have a class, and anything with a class is an object (formula 3).

Translation of Relations.

Similarly to objects, we use FOL domain elements to represent tuples (which are instances of relations). Therefore, in addition to some domain elements representing objects, some represent tuples. A convenient consequence of this approach is that it allows us to define creating and deleting of objects and tuples in a uniform way.

We introduce a new unary predicate $is_tuple(t)$ that returns true if the domain element t represents a tuple. We define that no domain element can be both an object and a tuple; their sets are mutually exclusive. For each relation $r \in R$ we introduce a unary predicate $r(t)$ that returns true if and only if t is representing a tuple that belongs to r.

In order to associate tuples with objects, for each relation $r \in R$ we define two additional binary predicates: $origin_r(t, o_1)$ and $target_r(t, o_2)$ which, together, identify that

$t = \langle r, o_1, o_2 \rangle$. Every tuple has exactly one origin object and one target object, and we enforce this condition using the following formula:

$$\forall t\colon r(t) \Rightarrow (\exists o_o\colon \text{origin}_r(t, o_o) \land \exists o_t\colon \text{target}_r(t, o_t) \land \\ (\forall o_1, o_2\colon (\text{origin}_r(t, o_1) \land \text{origin}_r(t, o_2)) \Rightarrow o_1 = o_2) \land \\ (\forall o_1, o_2\colon (\text{target}_r(t, o_1) \land \text{target}_r(t, o_2)) \Rightarrow o_1 = o_2))$$

We enforce cardinality constraints of r using similar formulae to limit the number of tuples per origin/target object.

Translation of Object Sets.

Object sets in ADS specifications are a fundamental component used in both actions and invariants, serving a purpose similar to expressions in conventional programming languages. An object set α represents a set of objects with a given, common superclass.

Every object set α is translated into a formula F_α that has two free variables x and o, denoted as $F_\alpha(x, o)$. F_α is true if and only if o belongs to the object set α under context x (we explain how contexts are used in Section 5.3 below). Object set formulae are meant to be injected into a formula G that quantifies x and o, producing a formula with no free variables assuming that G has no free variables.

For example, the `Subset` node in an ADS specification semantically evaluates to an object set that is a subset of its argument object set. To translate a `Subset` node, we introduce a new binary predicate $subset_\alpha(x, o)$ and enforce no additional constraints on this predicate. This predicate, if it holds on x and o, denotes that o will be chosen as part of this subset under context x. The resulting formula is:

$$F_{subset}(x, o) \equiv subset_\alpha(x, o) \land F_\alpha(x, o)$$

where $F_\alpha(x, o)$ is the translation of the argument object set for the `Subset` node (i.e., its child). The resulting formula still has free variables x and o, meaning that it is a valid object set formula. Notice that we enforced this subset function to be non-deterministic by not having any rules on which o is included in the subset using $subset_\alpha(x, o)$.

5.3 Translation of Statements

Actions in ADS are specified using sequences of statements. Each individual statement is translated into formulae that define the transition between the *pre-data store states* (the states before the statement is executed) and the *post-data store states* (the states after the statement is executed) (from now on, shortened to pre- and post-states). These formulae are designed to encapsulate the semantics of an individual statement independently from previous or successive statements. We can combine formulae for individual statements together by joining intermediate states: if a statement s_1 is followed by a statement s_2, we conjoin them by stating that the post-state of s_1 is equivalent to the pre-state of s_2. This method is used to combine statements into blocks, and ultimately entire actions.

Statements may be executed multiple times if they are contained within a loop body. In order to identify different executions of a statement, we introduce *contexts*. Like objects and tuples, contexts are represented using FOL domain elements. We use contexts to enforce the number of executions of each statement by enforcing the number of contexts for that statement. We create a unique context for the root block of any action in order to simulate exactly one execution of an action. Loops define their own sub-contexts in order to define executions of loop bodies. Therefore, every pre- and post-state s is represented by a predicate $instate_s(x, a)$ that denotes that object or tuple a is part of data store state s during the execution identified by context x.

$$\forall x, o\colon \text{is_object}(o) \Rightarrow (\\ \text{in_state}_{post}(x, o) \Leftrightarrow \text{in_state}_{pre}(x, o) \land \neg F_\alpha(x, o)) \quad (1)$$

$$\forall x, t\colon \text{is_tuple}(t) \Rightarrow \text{in_state}_{post}(x, t) \Leftrightarrow (\text{in_state}_{pre}(x, t) \\ \land \neg(\exists o\colon \text{in_state}_{pre}(x, o) \land F_\alpha(x, o) \land \text{origin}_{r_{o1}}(t, o)) \\ \cdots \land \neg(\exists o\colon \text{in_state}_{pre}(x, o) \land F_\alpha(x, o) \land \text{origin}_{r_{on_o}}(t, o)) \\ \land \neg(\exists o\colon \text{in_state}_{pre}(x, o) \land F_\alpha(x, o) \land \text{target}_{r_{t1}}(t, o)) \\ \cdots \land \neg(\exists o\colon \text{in_state}_{pre}(x, o) \land F_\alpha(x, o) \land \text{target}_{r_{tn_t}}(t, o))) \quad (2)$$

Figure 7: Formulae defining a delete statement

Below we detail the translation of several statements that demonstrate the methodology. The translation of other statements is omitted for brevity.

Translation of the Delete statement.

A `Delete` statement deletes all objects from a given object set. In addition it deletes all tuples that are related to a deleted object. Other objects and tuples are unaffected.

Given an object set α (as the argument of the `Delete` statement) and its translation into an open formula $F_\alpha(x, o)$, formulae in Figure 7 define the translation from the pre-state into the post-state of the delete statement, defined by predicates $\text{in_state}_{pre}(x, a)$ and $\text{in_state}_{post}(x, a)$ respectively.

Formula 1 in Figure 7 defines that an object belongs to the post-state if and only if it belonged to the pre-state and did not belong to the object set.

A tuple t in context x persists into the post-state if and only if it existed in the pre-state and t does not touch a deleted object (more precisely, if none of the predicates defining relations originating or targeting the object set class apply to t and a deleted object). Let c be the class of the object set, and let n_o and n_t be the numbers of relations that have c or its subclass as the origin or target class, respectively. We define r_{oi} to be a relation that has c or its subclass as the origin class, for $1 \leq i \leq n_o$. We analogously define r_{ti}. With those definitions in mind, Formula 2 in Figure 7 defines how a `Delete` statement migrates tuples between states.

Assignment statements and variables.

Before translation we convert each action into the static single assignment (SSA) form. Since after this transformation every variable v is assigned to by exactly one `Assign` statement s, assuming we have an object set α on the right hand side of the assignment, we can translate an assignment statement by introducing a binary predicate $v(x, o)$ and enforcing that $v(x, o)$ holds if and only if $F_\alpha(x, o)$.

The formula corresponding to the object set of a variable v is $v(x, o) \land \text{in_state}_{current}(x, o)$. The additional constraint of the object belonging to the current state is necessary because objects that were part of the variable may have been deleted since the assignment of this variable.

Translation of Loops.

There is only one type of loop in ADS language: the `ForEach` statement. It iterates through an object set, executing the loop body once for each object in the object set. The loop body can access the iterated object through the iterator variable. The order of iterated objects is non-deterministic.

The loop body needs to be executed once for each object in the supplied object set α. Since we use contexts to define executions, we need to define that, for each object in α under context x_p (the parent context), there exists exactly one context x_c (the child context) that will be used to execute the loop body. We specify this by using a predicate $context_link(x_c, o)$ denoting that we created context x_c to execute the loop body for an object o from the iterated object set. Child contexts are linked to their parent context using the predicate $parent_context(x_p, x_c)$.

We also define the pre- and post-state of the loop body, referred as pre- and post-step states. Each pre-step state is equal to the post-step state of the preceding iteration, with the exception of the first iteration whose pre-step state is equal to the pre-state of the `ForEach` statement. The `ForEach` statement's post-state is equal to the last post-step state.

Since we need to reason about iteration order, we introduce a predicate $prev(x_1, x_2)$ that defines that the iteration defined by context x_1 is executed before x_2's iteration, and that x_1 and x_2 belong to the same parent context x_p. Further constraints are added to impose a total order between all contexts with the same parent context.

Our experiments with the Spass theorem prover have shown that the theorem prover often cannot give conclusive results about actions with a `ForEach` statement based on the translation outlined above. The problem is that the theorem prover attempts to deduce all transitive formulae achievable by executing the iterations in a total order. This chain of deductions would terminate if the number of iterations is bounded (i.e., if the object set that is iterated upon is bounded). If the size of the iterated object set is unbounded the theorem prover never terminates, leading to an inconclusive result.

We developed an optimized translation that resolves this issue for a class of loops that are common in practice. We have observed that, in most loops that update the datastore, iterations do not affect each other (such as each iteration redefining a variable read by the following one, or creating objects that change the semantics of following iterations, etc.), i.e., there is no dependency between iterations of a loop. For such loops, it is unnecessary to define a total order on iterations, and we developed an optimized translation for `ForEach` loops based on this observation.

The optimized translation does not impose a total order between iterations and instead executes them concurrently. In this translation, entities (objects or tuples) exist in the post-state of the loop if and only if the entity existed in the pre-state of the loop and still exists in all post-step states, or if it did not exist in the pre-state of the loop and exists in at least one post-step state. The lack of total order among the iterations in this optimized translation makes the resulting FOL formulae more tractable. This optimized translation was applicable to all loops of the three applications we experimented on.

5.4 Translation of Invariants

Invariants are formulae described by a syntax very similar to FOL and are translated almost verbatim. Some additions are functions over object sets, quantification over object sets with quantified variables being singleton object sets etc.

For inductive invariant verification, all invariants should be assumed to hold before an action is executed; then we try to show that all invariants hold after the action execution. We add all invariants into the set of axioms, limiting each quantification and object set to cover only objects and tuples that exist in the pre-state of the action. We also add all invariants into the set of conjectures, this time limited such that all quantifications and object sets span objects and tuples that belong to the post-state of the action.

	FatFreeCRM	**Tracks**	**Kandan**
Lines of Code	30358	18023	2173
# ADS Nodes	85447	37755	907
# Nodes after optimization	1611	1483	280
# Classes	30	10	5
# Actions	167	70	35
# Invariants	8	10	5
# Empty actions	107	52	31
Avg. # of predicates	286	205	69
Theorem prover peak memory	243Mb	203Mb	126Mb
Avg. time per action/invariant	3.1 sec	40.5 sec	10.5 sec
# Action/invariant pairs	480	180	20
Verified	468	133	17
Falsified	2	2	0
Inconclusive	8	34	1
False positives	2	2	1
Detected Exceptions	0	9	1

Figure 8: Extraction and verification statistics

6. IMPLEMENTATION & EXPERIMENTS

Based on the techniques we discussed above, we implemented a fully automatic extraction and verification tool for Rails data models as a Rails gem (library) that can be obtained at `http://bocete.github.io/adsl/`. It is implemented in Ruby in 15689 lines of code and is licensed under the open LGPL license.

6.1 Experiments

We evaluated the tool on three Rails applications: FatFreeCRM, Tracks and Kandan. These applications are multiuser, have complex data models and are actively developed and maintained by the open source community. As a consequence of their open source nature these applications have been exposed to varying coding styles and standards, making them suitable to demonstrate the flexibility and limitations of our techniques. Statistics on these three applications can be found in Figure 8.

FatFreeCRM is an application for customer-relation management. The coding style is characterized by high flexibility in the model, where often the types and names of objects and associations are generated using string manipulations instead of static declarations. That way the same implementation of an action can be used for different controllers and actions. Because the extraction was done in instrumented runtime, the translator had sufficient information to automatically infer the correct semantics of these actions, which would not be doable by pure static analysis. Our tool found two bugs in FatFreeCRM that we reported to the developers, who confirmed and fixed one of them.

Tracks is an application for organizing tasks. It is interesting because of an unorthodox pattern in the data model, where the association among the Todo objects defines a partial order. This pattern posed a challenge for the verification stage. Our tool found two bugs in Tracks that we reported to the developers, who confirmed and fixed one.

Kandan is a chat application that heavily relies on third-party gems, most notably for authentication and authorization (devise[8] and cancan[4]). These libraries offer many features that are manually implemented in Tracks and Fat-

FreeCRM, and therefore, much Kandan's application logic is handled by external gems.

6.2 Extraction Details

Our tool initiates application verification by inspecting the application's exposed URL patterns, instrumenting corresponding actions and libraries and invoking requests, as described in Section 4. We manually entered invariants based on our inspection of the applications by using the Rails extension we developed for specifying invariants.

The process of automated extraction took up to 2 minutes for FatFreeCRM and significantly less for other applications. The extracted ADS specifications are very large (Figure 8) which results in very large formulas when translated to FOL. The automatically extracted specification can be significantly reduced using an ADS optimization phase that identifies redundant nodes such as assignments to variables that are never read, `Either` statements with empty or identical blocks, loops with empty bodies etc. Empty blocks are frequently encountered since operations on basic types are abstracted away during extraction. The optimization phase removes redundant nodes, and alters nodes in order to remove paths that do not modify the data store (as these paths trivially preserve all invariants). These optimizations reduce the size of the specification by an order of magnitude.

We manually checked each extracted ADS action against the actual code to ensure that the extracted specification is a sound abstraction of the implementation. We identified only one unsound action extraction in Tracks where a part of the action business logic is implemented inside the view. This is a violation of the MVC paradigm, and since our extraction approach relies on the MVC pattern, it is not surprising that we are unable to extract a sound abstraction in this case.

We found two interesting action extraction instances in Tracks. One is related to a coding error in the actual action. The action is deprecated but still public and would raise an exception if called. Our tool captured the exception during instrumentation which resulted in an empty action. This was the correct translation, but demonstrates that the extraction procedure is sensitive to runtime errors.

Another interesting example comes from one of the model methods being recursive (`Todo#touch_predecessors`). Since calls are inlined during extraction, this leads to an infinitely deep abstract syntax tree (until a stack overflow exception is raised during instrumented runtime). Since the operations inside this recursive call were abstracted away (they were only updating a field with a basic type), this entire call chain results in a no-op and is entirely removed during the optimization phase, resulting in a sound translation. Had there been updates to the objects and relations of the data model in this recursive call, the recursion would have been effectively unrolled a large but finite number of times, resulting in an unsound abstraction.

6.3 Verification Details and Identified Bugs

Our tool verifies each extracted ADS specification using the Spass[38] theorem prover. The tool first removes all empty actions from ADS as they trivially preserve all invariants. Many actions do not modify the data store and instead just look up data to show to the user; these actions are empty in ADS. Our tool then creates a FOL theorem for each remaining non-empty action-invariant pair, checking if, assuming that all invariants hold in the pre-state, the invariant holds in the post-state.

Spass verifies a FOL theorem by negating the conjecture and exhaustively exploring the space of deductible formulae. As soon as it deducts a contradiction, it has proven that the conjecture holds. To prove that the conjecture does not hold it needs to explore the entire space of deductible formulae and never deduct a contradiction, which takes much longer and may never terminate since FOL is undecidable.

For each action/invariant pair, our tool runs two instances of Spass concurrently with different heuristic settings; this increases the possibility of reaching a conclusive answer. The peak memory statistic in Figure 8 accounts for both Spass instances together. Our tool lets the theorem prover run for 3 minutes before interrupting it and classifying the result as inconclusive. We let the tool run for as much as 30 minutes for each action/invariant pair, and we did not find a single instance where a conclusive result was found in more than 2 minutes. Evaluations were done on a computer with an Intel Core i5-2400S processor, 32GB RAM, running 64bit Linux.

From a total of 480, 180, and 20 action/invariant pairs our tool proved 468, 133, and 17 action/invariant pairs correct in FatFreeCRM, Tracks and Kandan, respectively. We found 2 bugs in Tracks and 2 bugs in FatFreeCRM. We reported them to the developers and two of them have been fixed since, other two pending a response.

In Tracks, one bug we found showed that it is possible to orphan an instance of a `Dependent` class. This stems from the way the `ProjectController` deletes a `Project`; it cleans up all `Project`'s `Todos`, but does not clean up the `Dependencies` of deleted `Todos`. The other bug is caused when a `User` is deleted using the `UsersController`. All `Projects` of the `User` are deleted, but `Notes` of deleted `Projects` remain orphaned.

In FatFreeCRM, one of the bugs we found relates to `Permission` objects. These objects define permissions for either `User` or `Group` objects. Our tool has shown that it was possible to have a `Permission` without associated `User` or `Group` objects. The other bug relates to `Todo` objects being orphaned when a `User` is deleted.

There were 5 cases of false positives where our tool reported that an action broke an invariant but, when we manually inspected, we found out that the action preserved the invariant. The false positives are mainly due to the fact that our tool over approximates the action behavior since branches are converted to non-deterministic choices.

Our tool also identified 10 cases (9 in Tracks, 1 in Kandan) where the actions were implemented under the assumption that a look up operation always finds something in the database. When the searched object was not found, these actions would raise an exception and display a default error message to the user. Handling these cases as exceptions is not a good programming practice since failing to find an object that matches a query is not an exceptional behavior and should be handled by the normal program flow. While these actions will not result in invariant violations during runtime (since a raised exception will halt execution before the invariant is broken), they still point out programming problems and provide useful feedback to the programmer.

7. RELATED WORK

Nijjar et al. present techniques for analysis and verification of data models in Rails applications [30, 31, 29]. The data model used in Nijjar et al.'s work is a static model that

does not represent the actions that modify the data store states. The properties are checked with respect to association declarations without considering how they are updated via actions. Moreover, some assumptions used in constructing the static data model (such as assuming `belongs_to` associations have exactly one associated object) are not guaranteed to hold by the Rails semantics. In contrast, our model captures the exact behavior of a data store by modeling how actions update the data store states.

Near et al. [28] developed Rubicon, a web application verification tool that adds quantification to unit tests and translates tests into verifiable Alloy specifications using symbolic execution. Rubicon uses the Alloy Analyzer for bounded verification of generated specifications.

Both Tracks and Fat Free CRM were analyzed by the tool developed by Nijjar et al. and Rubicon. Neither of their tools were able to uncover the four bugs we found. Since the tool developed by Nijjar et al. only analyzes the static data model it is unable to capture the semantics of actions and find the bugs in them. Rubicon's approach, on the other hand, is a testing based framework. Successful tests verify that, after an explicitly stated sequence of steps, the application behaves as expected. Tests are not suitable for verifying that no possible sequence of action executions could lead to a faulty state, which is the condition checked by our verification framework.

Alloy [21, 22] is formal language for specifying object oriented data models and their properties. Alloy Analyzer is used to verify properties of Alloy specifications. Unlike our work, Alloy focuses on static models and does not directly support specification of actions or dynamic behavior. Moreover, Alloy Analyzer uses SAT-based bounded verification techniques as opposed to the FOL based unbounded verification technique used in this paper.

There has been prior work on formal modeling of web applications, mainly focusing on state machine based formalisms to capture the navigation behavior[36, 17, 2, 16, 39, 33]. In contrast to these previous efforts, we are focusing on analysis of the data model rather than the navigational aspects of the web applications.

There are previous results on unbounded verification of data-driven web applications based on high level specifications [6, 5, 7]. Deutsch et al. model actions as input/output rules instead of specifying them procedurally, creating a semantic gap between the implementation and the specification of the actions. Due to the semantic gap between the input/output rule format used in their language and the actual implementations of actions, the bugs we found would not be discovered by their verification approach. Additionally, they impose restrictions on the use of quantification in their properties whereas we do not have any restrictions.

Verification of software using theorem provers has been explored before in projects such as Boogie [1], Dafny [26] and ESC Java [11]. These projects focus on languages such as C, C#, and Java, and typically require user guidance in the form of explicit pre- and post-conditions, explicit data structure constraints, and loop invariants. Both our model and domain of application are significantly different. We focus on objects and associations that are changed by high-level code in web applications.

Another line of related work is inductive verification of abstract data type specifications [15, 27, 13], where algebraic specifications are used to model behaviors of data types such as stacks, queues etc., and automated verification techniques based on term-rewriting systems are used for verification. The types of specifications we focus on, and the verification techniques we use, are significantly different.

There has been work on the static analysis of Rails [19]. This line of work focuses on typechecking Rails applications and builds on DRuby [12], which is a Ruby static typechecker. The presented techniques infer types and detect errors by converting each statement into a type constraint, and exhaustively applying a set of rewrite rules. This is a different approach than ours and achieves a different goal, albeit on the same target language with certain common challenges (ghost methods and dynamic types).

Rubydust [20] attempts to typecheck Ruby code, accomplishing this by wrapping objects with type constraints and running actual code. They also use some basic instrumentation. This makes their approach similar to the core idea behind our extraction by instrumented execution technique, and in both cases, the goal is to get around dynamic features of Ruby. However, their wrapped objects contain type information whereas we inject abstract syntax trees into ActiveRecord objects and variables. Their solution focuses on general purpose Ruby which our solution cannot cover, but within our domain and with the code generation purpose, we cover much larger applications.

Finally, our verification approach has some high-level similarities with the long line of work in the area of symbolic execution [18, 24, 14, 34, 3]. For example, instrumentation has been used in symbolic execution for extraction [24, 3]. However, apart from this core idea of using runtime information to guide extraction, our goals and solutions largely differ. We extract a specification from a given application that represents all behaviors rather than extracting path conditions for specific paths as it is done in symbolic execution. We do not unroll loops. We generate FOL formulae with quantification, whereas path conditions extracted in symbolic execution are typically unquantified constraints that can be checked with SMT-solvers.

8. CONCLUSIONS AND FUTURE WORK

Cloud-based software applications store their data on remote servers and use data models to capture the interface between the back-end data store and the rest of the application. In this paper, we presented techniques for verification of actions that update the back-end data store in such applications. We achieve this by first automatically extracting a formal data model in our intermediate language. We show that invariants of the data model can be checked by translating this intermediate representation to FOL formulae and then using a FOL theorem prover. We implemented this approach and conducted experiments on three open source applications. We uncovered four previously unknown bugs two of which were acknowledged and fixed by the developers. There are several directions for future work. One of them is integration of other verification techniques, such as SAT-based bounded verification or SMT-based unbounded verification, to our framework. Bounded-verification can be used for the cases where the FOL theorem prover does not generate a conclusive result, whereas SMT-solvers can be used for verifying properties about object fields with basic types which are abstracted away in our current approach.

9. REFERENCES

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[2] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *Proceedings of the 24th IEEE/ACM International Conference Automated Software Engineering (ASE 2004)*, pages 100–109, 2004.

[3] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI 2008)*, pages 209–224, 2008.

[4] cancan | RubyGems.org | your community gem host, Sept. 2013. `http://rubygems.org/gems/cancan`.

[5] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *Journal of Computer and System Sciences*, 73(3):442–474, 2007.

[6] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. A system for specification and verification of interactive, data-driven web applications. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD Conference*, pages 772–774. ACM, 2006.

[7] A. Deutsch and V. Vianu. WAVE: Automatic verification of data-driven web services. *IEEE Data Engineering Bulletin*, 31(3):35–39, 2008.

[8] devise | RubyGems.org | your community gem host, Sept. 2013. `http://rubygems.org/gems/devise`.

[9] The Web framework for perfectionists with deadlines | Django, Feb. 2013. `http://www.djangoproject.com`.

[10] Fat Free CRM - Ruby on Rails-based open source CRM platform, Sept. 2013. `http://www.fatfreecrm.com`.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[12] M. Furr, J. hoon (David) An, J. S. Foster, and M. W. Hicks. Static type inference for ruby. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2009)*, pages 1859–1866, 2009.

[13] S. J. Garland and J. V. Guttag. Inductive methods for reasoning about abstract data types. In J. Ferrante and P. Mager, editors, *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 219–228. ACM Press, 1988.

[14] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 05)*, pages 213–223, 2005.

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

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

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

[18] S. L. Hantler and J. C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[19] J. hoon (David) An, A. Chaudhuri, and J. S. Foster. Static typing for ruby on rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 590–594, 2009.

[20] J. hoon (David) An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 459–472, 2011.

[21] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Enginnering and Methodology (TOSEM 2002)*, 11(2):256–290, 2002.

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

[23] kandanapp/kandan, Sept. 2013. `http://github.com/kandanapp/kandan`.

[24] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[25] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming (JOOP 1988)*, 1(3):26–49, Aug. 1988.

[26] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[27] D. R. Musser. On proving inductive properties of abstract data types. In *Proceedings of the 7th ACM Symp. Principles of Programming Languages (POPL 1980)*, pages 154–162, 1980.

[28] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *Proceedings of the ACM SIGSOFT 20th Int. Symp. Foundations of Software Engineering (FSE 2012)*, pages 60:1–60:11, 2012.

[29] J. Nijjar, I. Bocić, and T. Bultan. An integrated data model verifier with property templates. In *Proceedings*

of the ICSE Workshop on Formal Methods in Software Engineering (FormaliSE 2013), 2013.

[30] J. Nijjar and T. Bultan. Bounded verification of Ruby on Rails data models. In *Proceedings of the 20th Int. Symp. on Software Testing and Analysis (ISSTA 2011)*, pages 67–77, 2011.

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

[32] Ruby on Rails, Feb. 2013. `http://rubyonrails.org`.

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

[34] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 05)*, pages 263–272, 2005.

[35] Spring Framework | SpringSource.org, Feb. 2013. `http://www.springsource.org`.

[36] P. D. Stotts, R. Furuta, and C. R. Cabarrus. Hyperdocuments as automata: Verification of trace-based browsing properties by model checking. *ACM Transactions on Information Systems (TOIS 1998)*, 16(1):1–30, 1998.

[37] Tracks, Sept. 2013. `http://getontracks.org`.

[38] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *Proceedings of the 22nd Int. Conf. Automated Deduction (CADE 2009), LNCS 5663*, pages 140–145, 2009.

[39] 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.