Automated Software Engineering
https://doi.org/10.1007/s10515-018-0249-2

Inductive verification of data model invariants in web applications using first-order logic

Ivan Bocić[1] · **Tevfik Bultan**[2] · **Nicolás Rosner**[2]

Received: 22 December 2017 / Accepted: 18 November 2018

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 concern for these applications is the correctness of the actions that update the data store, which are triggered by user requests. Considering that modern applications store and manage data for millions (even billions) of users, misuse or loss of user data can have catastrophic consequences. In this paper, we focus on automated discovery of data store bugs in applications that use development frameworks that are RESTful, enforce the Model–View–Controller architecture, and use Object Relational Mapping libraries to manipulate data. We present a formal data model for data stores and data store manipulation in such applications. We have developed a framework for verification of data models via translation to First Order Logic (FOL), followed by automated theorem proving. Due to the undecidability of FOL, this automated approach does not always produce a conclusive answer. We investigate the use of many-sorted logic in data model verification in order to improve the effectiveness of this approach. Many-sorted logic allows us to specify type information explicitly, thus lightening the burden of reasoning about type information during theorem proving. Our experimental results demonstrate that our verification approach is scalable to real-world web applications and is able to detect bugs in them.

Keywords Data model · Automated verification · First-order logic · Many-sorted logic

This material is based on research sponsored by NSF under Grant CCF-1423623.

✉ Nicolás Rosner
rosner@cs.ucsb.edu

Extended author information available on the last page of the article

Published online: 12 December 2018

1 Introduction

Modern software applications have migrated from the desktop onto to the cloud. Benefits of web applications over desktop applications include accessibility on multiple devices anywhere and anytime, higher availability due to redundant systems, easier upgrades and patching etc. However, these benefits come at the cost of increased complexity, as web applications are complicated, distributed systems. Many web applications store and manage significant amounts of sensitive user data. Verification of how applications manage data is of paramount importance.

Modern software applications are challenging to develop and maintain due to the complexity of building software systems that consist of distributed components that run concurrently and interact over the Internet. In order to reduce this complexity and achieve modularity, web application development frameworks have been created for various languages: Ruby on Rails (2013) for Ruby, Django (2013) for Python, and Spring (2013) for Java. These frameworks share similar architectures and basic features shown in Fig. 1.

In particular, web application development frameworks use the Model–View–Controller (MVC) pattern (Krasner and Pope 1988) to separate the code for the model (Model) from the user interface logic (View) and the navigation logic (Controller). The data model defines the data the application manages, as well as the methods that are used to modify the data. The controller accepts requests, queries and/or updates the data, and invokes the view to synthesize the response. These operations are defined as *actions*. Web applications built using these frameworks have the following characteristics: (1) They are RESTful (Fielding 2000), meaning that actions can be invoked any number of times and in any order; (2) Actions are (or should be) atomic, meaning that they update the data in one step and revert any changes if an error is encountered; (3) The data is manipulated only by actions, meaning that there is no way to modify the data outside of actions; (4) The data model is implemented using Object-Relational Mapping (ORM) libraries that help bridge the semantic gap between object oriented languages and relational databases.

Although there are web applications that do not adhere to these design principles, popular open-source web application frameworks such as Ruby on Rails (2013),

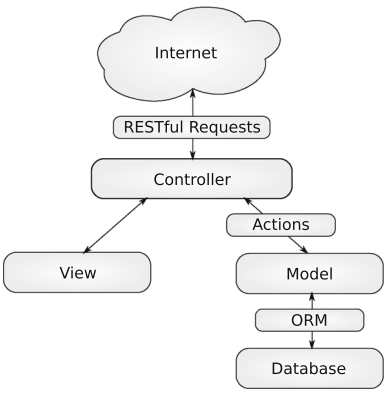

Fig. 1 Architecture imposed by web application frameworks

Django (2013), and Spring (2013) enforce them. Web applications that are based on these frameworks would benefit from the verification techniques we present in this paper.

We implemented our approach for Ruby on Rails, or Rails in short. We decided to focus on the Rails framework since it is widely used. However, our approach can be adapted to other MVC-based web application frameworks such as Django and Spring. As of October 2018, SimilarTech (2018) reports 337,569 public websites built using Rails, 62,681 using Django, and 3378 using Spring. (The actual number of Spring applications is likely higher, as Spring is often used in intranets and business-to-business applications.)

The aforementioned characteristics and the modularity induced by the web application development frameworks enabled us to develop a framework for automated verification of data model invariants by analyzing the object oriented code that defines the actions, and verifying that each action preserves the invariants. Our key observation is that, in RESTful applications, actions are atomic and can be executed in any order. This lets us use inductive verification to verify data integrity by considering each action in isolation and checking whether an action could possibly invalidate a property that was presumed to be valid before the action executed. If no action breaks any property, then assuming the application starts executing in a valid state, no invalid state could possibly be reached.

Our framework for verification of web application data models automatically translates verification queries to logic formulas and then uses an automated theorem prover to check these automatically generated formulas (Bocic and Bultan 2014, 2015c). First, by exploiting the structure of the MVC-pattern, we automatically extract a formal data model, which we call *Abstract Data Store (ADS)*. ADS models capture the semantics of the database schema and the actions that update the data store (Bocic and Bultan 2014). ADS models represent the data store as sets of objects (corresponding to objects of the data model classes) and associations among them (corresponding to the associations among the data model classes). Attributes that correspond to basic types are not represented in ADS models (i.e., they are abstracted away). This means that we can verify invariants about sets of objects and associations among them, but for example, not about numeric attributes of objects.

We statically extract the abstract data store model from the given web application by analyzing the data model schema and the methods that implement the actions (Bocic and Bultan 2017). We ask the user to write data model invariants to be verified using our invariant specification library that provides constructs for quantification. For each action-invariant pair, we synthesize a first order logic (FOL) theorem that is valid if and only if the action preserves the invariant. Assuming that the invariant is true before the action starts executing, and specifying the way the action modifies the data, the theorem posits that the invariant must hold after the action ends its execution. We send this theorem to an off-the-shelf FOL theorem prover to verify.

Since FOL is undecidable, an automated theorem prover may never terminate deducing, continuously producing new deductions without reaching a proof of the theorem. Our approach results in one of three outcomes for each action-invariant pair: (1) a proof that the action preserves the invariant, (2) a proof that the action can violate the invariant, or (3) an inconclusive result, caused

by the theorem prover not reaching a conclusive answer in a specified time period.

Minimizing the ratio of inconclusive results is a necessary step for making our approach usable in practice. In order to understand the causes of inconclusive results, we investigated the logs of the first theorem prover we integrated to our framework (Spass Weidenbach et al. 2009). We noticed that the theorem prover performed an excessive number of deductions solely to reason about the types of quantified variables and objects. Since FOL does not have a notion of type, our FOL translation generates predicates that encode all the type information, and the theorem prover was spending a lot of time making deductions about these predicates.

In order to address this problem, we looked into using many-sorted logic for data model verification. In many-sorted logic, sorts (i.e., types) are explicitly associated with all variables, functions and predicates. Our intuition was that using sorts will benefit verification because it mitigates the necessity of deducing type information. On the other hand, the semantics of sorts and the semantics of data model classes do not match, and this semantic mismatch makes the translation of data models challenging especially if inheritance is present in the data model.

In addition to dealing with inheritance, there is one more complication in translation to many-sorted logic. Classical FOL (i.e., FOL without sorts) defines structures (instances that may or may not satisfy a given set of formulas) as strictly non-empty. In our data model encoding, empty structures represent the possibility of a particular class having no objects, and, we want to allow empty structures since this might be a possible behavior of the data model. Unfortunately, most theorem provers do not allow empty structures. This problem was not a significant issue with our translation to unsorted FOL since we were encoding types with predicates, and it is possible to define a predicate that never evaluates to true (which would encode an empty class). However, when we map classes to sorts, the issue of empty structures must be handled during translation which introduces extra complexity, further distancing sorts from the data model type system.

We developed a data model to many-sorted logic translator that addresses these issues. As a many-sorted logic theorem prover we used Z3 (de Moura and Bjørner 2008), a Satisfiability Modulo Theories (SMT) (SMT-LIB) solver. SMT problems are problems that can be expressed in FOL with equality and different background theories such as linear arithmetic, bitvectors etc. When used with uninterpreted functions and quantification only, SMT provers can be used as general purpose FOL theorem provers.

In order to compare the performance of the unsorted FOL (i.e., FOL without sorts) translation with the many-sorted logic translation we used Spass and Z3. We compared the performance between Spass (using the unsorted FOL translation) and Z3 (using the many-sorted logic translation) by extracting action-invariant pairs from 17 open source web applications, and translating them to formulas for Spass and Z3. We observed that verification results for Z3 were significantly better. We found that Z3 outperformed Spass, producing far fewer inconclusive results. In addition, we observed a speedup of two orders of magnitude over Spass. This performance difference was beyond our expectations.

However, looking just at these results, it is not possible to attribute the performance improvement to the benefits of many-sorted logic translation over the unsorted FOL

translation. Spass and Z3 use different deduction methods, which could be the cause of the performance difference. Or, the performance difference could even be due to differences in the implementations and optimizations of the different provers.

In order to determine the cause of the performance difference we developed a translation of the data model to many-sorted logic that effectively bypasses the sort system. We re-ran our experiment suite on Z3 with this unsorted translation. We found that the unsorted translation induced an order of magnitude higher inconclusive result rate, and slowed down verification by two orders of magnitude. While Z3 and Spass use fundamentally different approaches to theorem proving, we conclude that sorts are inherently useful in data model verification.

To experimentally evaluate our overall approach, we applied our automated verification framework to 17 open source web applications and checked 17,291 action-invariant pairs. Of these 17,291 verification conditions, our framework verified 17,133 to be correct. For the remaining 158 verification conditions: (1) For 20 of them the verification process was inconclusive since our theorem provers did not produce a result within the allocated time. (2) For 45 of them our verification approach reported that the property fails, however, after manual investigation we observed that these were false positives corresponding to infeasible behaviors (which is possible since the ADS model we use abstracts the behavior of the application, hence, can include infeasible behaviors). (3) For 93 properties our verification approach reported that the property fails and we manually confirmed that these are true positives where the property failure exists in the application. In fact, we identified 69 unique bugs in 17 applications that were exposed by these property violations.

The rest of the paper is organized as follows. Section 2 discusses the formal data model we developed as part of our formal verification framework, Sect. 3 explains the translation of verification queries about the data models to first order logic. Section 4 discusses verification via many-sorted logic. Section 5 presents experiments evaluating the effectiveness of different logic encodings and the overall effectiveness of the proposed verification approach. Section 6 discusses related work, and Sect. 7 concludes the paper.

2 Abstract data stores

In this section we define abstract data stores (ADSs), a formal model we use for representing data models of web applications (Bocic and Bultan 2014). We use excerpts from an application as a running example first to demonstrate the data models in Rails and then to demonstrate how ADSs capture data models of Rails applications. In addition, ADSs can capture properties (invariants) defined on data models.

2.1 Data models in Ruby on Rails

Tracks (2013) is a Rails application for organizing tasks, to-do lists etc. This application spans 17,562 lines of code, 11 model classes and 117 actions. Figure 2 an example excerpt from Tracks. This excerpt would normally be contained in multiple files, one

```
1  class User
2    has_many :todos
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 = current_user
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
```

Fig. 2 Excerpt from a Rails application

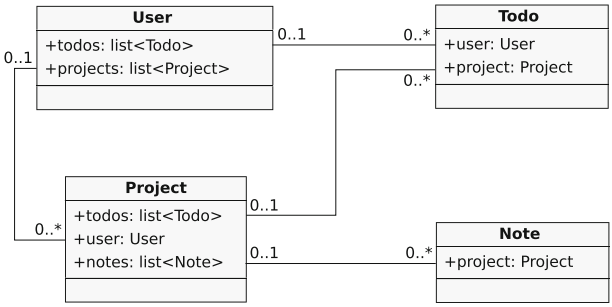

Fig. 3 Class diagram corresponding to Fig. 2

for each model class and one for each controller. For brevity, we only show relevant details in Fig. 2.

Lines 1–16 in Fig. 2 demonstrate how ActiveRecord (default ORM for Ruby on Rails) can be used to define a data model. The example application defines four ActiveRecord classes: `User`, `Project`, `Todo` and `Note`, declared in lines 1–4, 5–9, 10–13 and 14–16 respectively.

Each class contains a set of associations (relations) with other classes. These associations are declared using methods `belongs_to`, `has_one`, `has_many` and `has_and_belongs_to_many` that imply different cardinality and schema details. Figure 3 shows the class diagram corresponding to the code given in Fig. 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, `Project.todos` and `Todo.project` are symmetrical: for every `Project` p and every `Todo` t of that `Project`, the `Project` of t is p.

Two actions can be seen in Fig. 2: one in `TodosController` called `create` (lines 18–26) and one in `ProjectsController` called `destroy`(lines 29–36).

The `TodosController#create` action takes an argument as part of the request, called `project_id`. This argument is used to lookup the corresponding `Project` object and assign it to a variable `@project` (line 19). In line 20, the current user object is stored in a variable called `@user`. The action, then, creates a new `Todo` instance (line 21), associates it with the loaded user and project 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`. The corresponding `Project` object is loaded in line 30 and stored in a variable. In line 31, the action iterates through all `Note`s associated with that project and deletes them one at a time (line 32). Finally, said project gets deleted (line 34).

Assume that we would like to verify the following property for the application in Fig. 2: *Each* `Todo` *object is associated with a* `Project` *object*. In order to do that, we first need a way to express this property. We developed a Rails library for specification of data model invariants using Rails syntax. For example, this property would be stated as:

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

For this property, our tool would show that the `TodosController#create` action preserves the given invariant, whereas the `ProjectsController#destroy` action potentially violates the invariant. If the deleted project had `Todo` objects associated with it at the beginning of the action, after deleting it, these `Todo` objects will be left with no associated `Project`, invalidating the invariant.

2.2 A formal data model: abstract data stores

In this section we define the formal data model that we use to capture the semantics of data models in web applications. The *abstract data store*, or *data store* in short, is an abstraction of a web application's data model that focuses on the persistent data that the application manages.

An abstract data store is a structure $DS = \langle C, L, A, I \rangle$ where C is a set of classes, L is a set of associations, A is a set of actions, and I is a set of invariants.

A *data store state* is a tuple $\langle O, T \rangle$ where O is the set of objects and T is the set of tuples denoting associations among objects. We define $\overline{DS}$ to be the set of all data store states of DS.

2.2.1 Classes and objects

The set of classes C identifies the types of objects that can be stored in the data store. Each class can have a set of superclasses ($\text{superclass}(c) \subset C$) and, transitively, the superclass relation cannot contain cycles. We will use operator $c_c < c_p$ to denote that c_p is a parent class to c_c, transitively or directly. We will use operators $>$, $\leq$ and $\geq$ accordingly.

For example, given the application presented in Fig. 2, C would encompass four classes: `User`, `Project`, `Todo` and `Note`. The superclass set of each of these classes is empty.

Given a data store state $\langle O, T \rangle \in \overline{DS}$, O is the set of objects that are stored in a data store in that state. Each object $o \in O$ is an instance of a class $c \in C$ denoted by $c = \text{classof}(o)$. We use the notation O_c to encapsulate all objects in O whose class is c or any subclass of c. We define $\overline{O}$ to be the set of all sets of objects that appear in $\overline{DS}$.

2.2.2 Associations and tuples

An association $l = \langle name, c_o, c_t, card \rangle \in L$ contains a unique identifier $name$, an origin class $c_o \in C$, a target class $c_t \in C$ and a cardinality constraint $card$. Cardinality constraints supported by ORM tools are limited, and so is our definition of valid cardinality constraints. Cardinality constraints are a pair of ranges n_o and n_t written as n_o-n_t. Ranges n_o and n_t describe the allowed number of objects on the origin and target side of the association, respectfully. The possible ranges are: $[0 \ldots 1]$, 1, $[1 \ldots *]$ and $*$. For example, cardinality constraint $1 - *$ defines that every target object is associated with exactly one origin object. Alternatively, cardinality constraint $[0 \ldots 1] - 1$ defines that every object of the target class is associated with an object of the origin class, and that no object of the origin class is associated with more than one object of the target class.

For example, given the application presented in Fig. 2, there are four associations in L:

$$l_1 = \langle \texttt{User\_todos}, \texttt{User}, \texttt{Todo}, 1 - * \rangle$$
$$l_2 = \langle \texttt{User\_projects}, \texttt{User}, \texttt{Project}, 1 - * \rangle$$
$$l_3 = \langle \texttt{Project\_todos}, \texttt{Project}, \texttt{Todo}, 1 - * \rangle$$
$$l_4 = \langle \texttt{Project\_notes}, \texttt{Project}, \texttt{Note}, 1 - * \rangle$$

Similar to how objects are instances of classes, tuples are instances of associations. Each tuple $t \in T$ is in the form $t = \langle l, o_o, o_t \rangle$ where $l = \langle name, c_o, c_t, card \rangle \in L$ and $\text{classof}(o_o) \leq c_o$ and $\text{classof}(o_t) \leq c_t$. 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.

Note that we did not define that data store states need to have association cardinality correctly enforced. This is because, sometimes, an action will temporarily invalidate cardinality while mutating data. In fact, cardinality is enforced when the data gets sent to the database, either by application-level validations or by the database schema directly. If cardinality constraints are violated, the action should abort without modifications to the data, trivially preserving all invariants. Hence, we treat cardinality constraints as implicit invariants that are necessarily correct before and after an action executes.

2.2.3 Actions

Given a data store $DS = \langle C, L, A, I \rangle$, A denotes the set of actions. Actions are used to query or update the data store state. Each action $a \in A$ is a set of *executions* $\langle s, s' \rangle \subseteq \overline{DS} \times \overline{DS}$ where $s = \langle O, T \rangle$ is the pre-state of the execution and $s' = \langle O', T' \rangle$ is the post-state of the execution.

In a web application implementation of an action is not specified as state transitions. Each action is implemented as a sequence of statements. However, each statement can be modeled as a state transition represented by a combination of boolean and object set expressions (i.e., expressions that return boolean values or a set of objects), and the state transitions for an action can be obtained by composing the state transitions of the statements used in the implementation of the action. In our verification framework, we use an intermediate representation called the Abstract Data Store Language (ADSL) to capture the semantics of action implementations in web applications (Bocic 2016). We give an overview of this representation in the next section.

2.2.4 Extraction of abstract data store models from web applications

Our verification framework requires extraction of an abstract data store model for a given web application. We do this by using the symbolic model extraction technique (Bocic and Bultan 2017) which is an automated technique for extracting formal data models from web applications. The description of the symbolic model extraction technique we use can be found in Bocic and Bultan (2017).

Given a web application, symbolic model extraction generates an Abstract Data Store Language (ADSL) specification characterizing the data model of the given application. We summarize some features of ADSL below, but we omit the full description of the ADSL here [which can be found in Bocic (2016)]. We note that the semantics of an ADSL specification corresponds to an ADS.

ADSL supports the specification of classes, inheritance relationships among classes, associations, and cardinalities of associations. The most involved part of an ADSL specification is the action specifications. Action specifications in ADSL consist of statements, boolean expressions (that evaluate to *true* or *false*) and object set expressions (that evaluate to a set of objects).

For example, a `Block` statement in ADSL corresponds to sequential composition of arbitrary number of statements. Each statement may migrate the data store state, or evaluate to a boolean or set of objects, or both. For example, an `Assign` statement in ADSL assigns the result of an object set expression to a variable, and the statement evaluates to the assigned object set.

Statements that return an object set evaluate to a set of objects in the data store that share a common class or superclass. Statements that return a boolean evaluate to *true* or *false*, typically used as conditions in branches.

Let us also give some examples of object set expressions in ADSL. The `AllOf(class)` expression returns all objects of class (or subclass of) `class`. On the other hand, the `Subset(object set)` expression returns an arbitrary subset (non-deterministically selected) of its argument object set. Similarly, the `OneOf(object set)` expression represents a non-deterministic selection of one object from its argument object set. Finally, `VarRead` expression takes a variable as an argument and evaluates to the set of objects assigned to that variable.

The complete description of the ADSL is provided in Bocic (2016). Note that ADSL is an intermediate representation used in our verification framework. It is automatically extracted from a given web application using the symbolic model extraction technique

(Bocic and Bultan 2017). In the following sections we discuss how we generate logic formulas from ADSL specifications for data model verification.

2.2.5 Invariants

Given a data store $DS = \langle C, L, A, I \rangle$, I is the set of invariants. An invariant $i \in I$ corresponds to a function $i : \overline{DS} \rightarrow \{\text{false}, \text{true}\}$ that identifies the set of data store states which satisfy the invariant. As we mentioned above, invariants can be specified using the Rails library we developed for specification of data model invariants.

2.2.6 Behaviors

A *behavior* of a data store is an infinite sequence of data store states such that the initial state satisfies all invariants, and each pair of consecutive states is covered by at least one action. Formally, given a data store $DS = \langle C, L, A, I \rangle$, a behavior of a data store 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
- $\forall i \in I : i(\langle O_0, T_0 \rangle) = \text{true}$

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

2.3 Data store correctness

If a data store preserves its invariants, then we say that it satisfies data integrity. Our goal is to verify the data integrity property for a given data store. Formally, given an abstract data store $DS = \langle C, L, A, I \rangle$, 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 $i \in I, i(\langle O, T \rangle) = \text{true}$. The verification problem for data integrity is 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 not a decidable 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 $DS = \langle C, L, A, I \rangle$ is consistent if the conjunction of all the invariants $i \in I$ is an inductive invariant. In other words, an abstract data store $DS = \langle C, L, A, I \rangle$ is consistent if and only if every execution of every action preserves all invariants:

$$F_{cons} \equiv \forall a \in A : \forall \langle s, s', \alpha \rangle \in a : (\forall i \in I : i(s)) \Rightarrow (\forall i \in I : i(s'))$$

3 Verification via first order logic

In this section we present the translation of ADS language specifications to classical first order logic and show how this translation can be used to verify whether invariants hold on a given ADS, i.e., if the given ADS is consistent and satisfies data integrity.

A FOL *language* L is a tuple $\langle F, P, V \rangle$ where F is a set of *function* symbols, P is a set of *predicate* symbols, V is a set of *variable* symbols. All function and predicate symbols are associated with their *arities*, which are positive integers denoting the number of arguments they accept.[1]

Given a FOL language $L = \langle F, P, V \rangle$, a *term* is a variable $v \in V$ or a function invocation $f(t_1, t_2 \dots t_k)$ where $f \in F$ and $t_1 \dots t_k$ are terms and k is the arity of function f.

A (well formed) FOL *formula* is defined as either:

- $p(t_1, \dots t_k)$, where $p \in P$ is a predicate of arity k and $t_1 \dots t_k$ are terms
- $\forall v : f$, where $v \in V$ and f is a formula
- $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ where f_1 and f_2 are formulas
- $t_1 = t_2$, where t_1 and t_2 are terms.[2]

Given a FOL language L, a *structure* S is an instance that may or may not satisfy a formula expressed in this language. More formally, it is a tuple $\langle U, F^S, P^S, V^S \rangle$ where U is a non-empty set of elements called the *universe*. F^S is a mapping of F onto a set of functions such that for every $f \in F$ of cardinality k there exists an $f^S \in F^S$ such that f^S is a function that maps $U^k \to U$. Similarly, for every predicate $p \in P$ of arity k, there exists a $p^U \in P^U$ such that $p^U \subset U^k$.

We can test whether a structure S satisfies a formula (whether the formula is *true* within this structure). To do this we assign elements of U to all terms in the formula. Each variable $v \in V$ is assigned an element $v^S \in U$. Term $f(t_1 \dots t_k)$ is mapped to the return value of f^U when using elements of U assigned to terms $t_1 \dots t_k$ as arguments. Similarly, $p(t_1, \dots t_k)$ is considered to be true if and only if elements corresponding to $t_1 \dots t_k$ form a tuple that is in P^U. Boolean operators and equality are interpreted in a standard way. Universal quantification is a bit more involved: $\forall v : f$ is satisfied by S if and only if, for every structure $S_{(v|e)}$ that is identical to S except that v was assigned a (potentially different) element e of U, f is satisfied by $S_{(v|e)}$.

A formula that is satisfied by one structure may not be satisfied by another. For example, $x = y$ is true for all structures that happen to map variables x and y to the same element. A formula $\forall x : (\forall y : x = y)$ is true if and only if U is a singleton set. If a formula is satisfied by all structures, we call this formula *valid*. E.g. $x = x$ is a valid formula.

[1] We may extend this to introduce constants as functions of arity 0 and propositional variables as predicates of arity 0.

[2] Although classical FOL does not include equality, since the theorem provers we use operate on FOL with equality, we include equality in our definition of FOL.

We take note of *free variables*: variables that are not quantified outside the term in which they appear. For example, $\forall x : x = y$ has one free variable y. Since theorem provers we use do not allow free variables, from this point on, we will only evaluate the truth value of formulas without free variables. Such a formula is true if and only if it is valid for all structures.

3.1 Translation of abstract data stores to first order logic

To translate an ADS to first order logic for verification, we create a different set of formulas for each action and for each invariant. We translate the schema (Sect. 3.1.1), the action (Sect. 3.1.5), and invariants in the pre-state (Sect. 3.1.6) into axioms. We also translate the invariant in question into a conjecture. If the resulting set of axioms implies the conjecture, then the action correctly preserves the invariant. If axioms do not imply the conjecture, a bug is reported as there exists a way for the action to invalidate the invariant that is being verified.

A single translation that models all actions and verifies all invariants at once is feasible, but we decided to not take this approach for two reasons. First, this would make identifying a detected bug difficult, as the theorem prover would show that an action could break an invariant without specifying which invariant and which action are the violators. Second, the resulting set of formulas would be rather large and if a theorem prover were not able to terminate for any isolated action/invariant pair, it would probably not terminate if given all actions and invariants (depending on theorem prover heuristics, it may be possible that this would instead terminate, though extremely unlikely). Such a failure would provide no partial result to the developer. By partitioning the problem and verifying each action/invariant property in isolation, the developer can get results for everything successfully proven or falsified even if there exist action/invariant pairs for which the theorem prover produced no conclusive result.

In this section we frequently conjoin or disjoin a set of formulas. When a set of conjoined or disjointed formulas is empty, we substitute the conjunction or disjunction with their neutral elements (*true* and *false* respectively).

In Fig. 4 we show the the class diagram that corresponds to a small part of the data model of a web application called FatFreeCRM. FatFreeCRM is an application for customer-relation management. It supports storing and managing customer data, leads that may potentially become customers, contacts, campaigns for marketing etc. It spans 20,178 lines of Ruby code, 32 model classes and 120 actions. We will use the class diagram shown in Fig. 4 as a running example in this section.

3.1.1 Schema translation

We assume that we are given a data store $DS = \langle C, L, A, I \rangle$.

Class translation First, for each class $c \in C$, we define a unary predicate $\overline{c}$ that semantically denotes whether its argument represents either an instance of that particular class or an instance of any subclass.

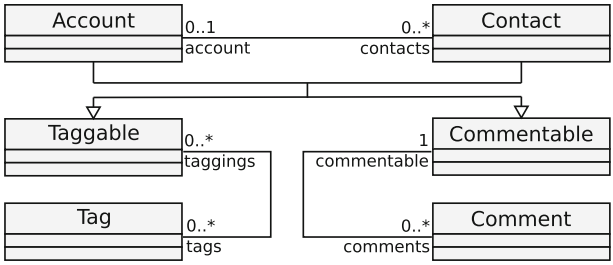

Fig. 4 A data model schema example based on FatFreeCRM (2013)

Predicates: `Account`, `Contact`, `Taggable`, `Commentable`, `Tag`, `Comment`, `XTaggable`, `XCommentable`.

$$\forall o\colon \texttt{Account}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o) \quad (1)$$

$$\forall o\colon \texttt{Contact}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o) \quad (2)$$

$$\forall o\colon \texttt{XTaggable}(o) \leftrightarrow \texttt{Taggable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \quad (3)$$

$$\forall o\colon \texttt{XCommentable}(o) \leftrightarrow \texttt{Commentable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \quad (4)$$

$$\forall o\colon \texttt{Account}(o) \rightarrow \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o) \quad (5)$$

$$\forall o\colon \texttt{Contact}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o) \quad (6)$$

$$\forall o\colon \texttt{XTaggable}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o) \quad (7)$$

$$\forall o\colon \texttt{XCommentable}(o) \rightarrow \neg\texttt{Account}(o) \vee \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o) \quad (8)$$

$$\forall o\colon \texttt{Tag}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Comment}(o) \quad (9)$$

$$\forall o\colon \texttt{Comment}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \quad (10)$$

Fig. 5 Axioms defining the class diagram in Fig. 4 in classical FOL

Then we define axioms that enforce our type system. We define three groups of axioms: *inheritance axioms* that define superclass relationships, *instance axioms* that define predicates that we can use to denote that an object is an instance of a given class (specifically not of a subclass), and *membership axioms* that define that every object is an instance of at most one class.

Inheritance axioms define that objects of subclass types are also of superclass types. For each class $c \in C$ that has a non-empty superclass set $\text{superclass}(c) = \{p_1, p_2 \ldots p_k\}$ we generate an axiom:

$$\forall o\colon \overline{c}(o) \rightarrow \overline{p_1}(o) \wedge \overline{p_2}(o) \wedge \cdots \wedge \overline{p_k}(o)$$

For example, given the model in Fig. 4 this method produces Formulas (1) and (2) in Fig. 5.

Instance axioms constitute one axiom per class $c \in C$ and serve to define *instance predicates* $\overline{c}_x$, where the x stands for the fact that c is the *exact* class of the corresponding object (denoted as the prefix `X` in the formulas in Fig. 5). More precisely,

these predicates are used to express that an object is an instance of class c, but not of any of c's subclasses. Given $\{s_1 \dots s_k\}$, the set of all direct subclasses of c (all classes s for which $c \in \text{superclass}(s)$), we generate an axiom:

$$\forall o : \overline{c}_x(o) \leftrightarrow \overline{c}(o) \wedge \neg\overline{s_1}(o) \wedge \dots \wedge \neg\overline{s_k}(o)$$

Note that, if c has no subclasses, this axiom defines equivalence between $\overline{c}$ and $\overline{c}_x$. If this is the case, as an optimization, we omit defining $\overline{c}_x$ and use $\overline{c}$ instead. Given the model in Fig. 4 this creates Formulas (3) and (4) in Fig. 5.

Membership axioms define that each object represents an instance of exactly one class. Assuming that $C = \{c_1 \dots c_k\}$, for every class $c_i \in C$, we create an axiom in order to constrain that, if an object is an instance of class c_i, it cannot be an instance of any other class:

$$\forall o : \overline{c_i}_x(o) \rightarrow \neg\overline{c_1}_x(o) \wedge \dots \wedge \neg\overline{c_{i-1}}_x(o) \wedge \neg\overline{c_{i+1}}_x(o) \wedge \dots \wedge \neg\overline{c_k}_x(o)$$

These formulas correspond to Formulas (5)–(10) in Fig. 5.

The resulting number of generated formulas is linear in the number of classes, and so is the size of these formulas.

Association translation Similarly to objects, we use FOL universe elements to represent tuples. A convenient consequence of this approach is that it allows us to define the creation and deletion of objects and tuples uniformly. We introduce unary predicates is_object and is_tuple to distinguish whether a universe element represents an object or a tuple. We define that no domain element can be both an object and a tuple.

For each association $l \in L$ we introduce a unary predicate $l(t)$ that returns true if and only if t is representing a tuple that belongs to l.

In order to associate tuples with objects, for each association $l \in L$ we define two unary functions: $origin_l(t)$ and $target_l(t)$ such that: $t = \langle l, origin_l(t), target_l(t) \rangle$.

We enforce association cardinality constraints using formulas to limit the number of tuples per origin/target object in a data store state. Note that we do not enforce cardinality globally, but only in the action's pre and post state. We do this because real world applications often invalidate cardinality temporarily while an action is executing.

3.1.2 Action translation

Actions are the most complex part of ADS translation to FOL. We will first define how states are represented in FOL, then define how object set and boolean expressions in ADSL are translated to FOL. For brevity, we will not discuss translation of all expressions in ADSL to FOL, and instead discuss translations of most representative operators used in ADSL expressions. Description of all ADSL expressions are provided in Bocic (2016).

States are translated to unary predicates that define which objects and tuples exist in the state. For example, given a state predicate s, if $s(x)$ then x is a domain element representing either an object or a tuple that exists in state s.

3.1.3 Object set translation

Most ADSL statements manipulate or return object sets. An object set α represents a set of objects that share a common set of classes or superclasses.

Every object set α is translated into a formula F_α that has one free variable o. Object sets inside loops are translated to formulas with more than one free variable, but for simplicity, we will focus on object sets outside loops in the following discussion.

F_α evaluates to true if and only if the free variable o is assigned a universe element that represents an object that belongs to the object set α. Object set formulas are meant to be directly injected into formulas that quantify over these free variables, and use the object set according to the statement's semantics.

For example, let us translate an object set expression `AllOf(c)` to FOL, where the `AllOf(c)` expression in ADSL evaluates to all objects of the class c. Let c be the type predicate corresponding to the argument class, and let s be a predicate denoting the state in which the object set is being evaluated. Then, the object set is defined simply using the formula:

$$s(o) \wedge c(o)$$

meaning that domain element o represents a member of this object set if and only if it exists in the current state s and is of class c. Note that o is the one free variable in this formula.

As an another example, a `Subset` expression in ADSL semantically evaluates to an object set that is a subset of its argument object set. Let β be the argument object set of a subset node α. To translate a `Subset` expression in state s, we introduce a new predicate $subset_\alpha(x)$ and define an axiom:

$$\forall x : subset_\alpha(x) \Rightarrow F_\beta(x)$$

With this axiom, $F_\alpha(o)$ is translated as: $subset_\alpha(o)$

The resulting formula still has o as a free variable, 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 $subset_\alpha(o)$.

Other interesting examples are `Assign` and `VarRead` expressions. Note that this translation is done after transforming the program to static single assignment that ensures that all variables are defined once.

The `Assign` expression takes two arguments: a variable identifier v and object set node α. It defines a predicate v that corresponds to the object set set α. It accomplishes this by defining an axiom:

$$\forall x : v(x) \Leftrightarrow \alpha(x)$$

After defining this predicate, the `Assign` expression returns $v(o)$ as its object set formula.

Variable read expression `VarRead` takes a variable v as an argument. When evaluated in state s, it translates to the object set formula denoting all object that were assigned to v and still exist in the current state:

$$v(x) \wedge s(x)$$

3.1.4 Boolean expression translation

Boolean expressions are translated to formulas that are embedded into statements, similarly to object sets. Unlike object set formulas, these formulas do not have free variables, unless inside a loop. Free variables are only introduced to facilitate loops, as will be explained further down in this section.

For example, the `IsEmpty` expression takes an object set as an argument and evaluates to $true$ if and only if the argument object set is empty. If object set α is this argument, the node translates to formula:

$$\forall x : \neg\alpha(x)$$

As one more example, the $\subset$ expression accepts two object sets and is expected to evaluate to $true$ if and only if the first object set is a superset of the other. Let $F_1(o)$ and $F_2(o)$ be the translations of the two object sets. The $\subset$ expression translates to the formula:

$$\forall o : F_1(o) \rightarrow F_2(o)$$

3.1.5 Translation of state migrations

Many ADSL expressions migrate the data store state. Each state migration can be represented as a set of pairs of states $\langle s, s' \rangle \subseteq \overline{DS} \times \overline{DS}$ which semantically represent possible state transitions by means of that statement. For a statement S and two states s and s', we will use $[s, s']_S$ to denote that $\langle s, s' \rangle$ is a possible execution (state transition) of S.

For example, let us translate a `Delete` statement D that deletes objects from an object set α, as well as all tuples associated with deleted objects. Assuming there exists only one association between class t and other types, this statement can be translated as follows:

$$\begin{aligned}
&\forall s, s' \in \overline{DS} \times \overline{DS} : \\
&\quad [s, s']_D \Leftrightarrow (\forall o : \text{is_object}(o) \Rightarrow (o \in s' \Leftrightarrow o \in s \wedge \neg F_\alpha(o))) \\
&\quad \wedge (\forall t : \text{is_tuple}(t) \Rightarrow t \in s' \Leftrightarrow (t \in s \wedge \neg(F_\alpha(\text{origin}_r(t)) \vee F_\alpha(\text{target}_r(t)))))
\end{aligned}$$

As an other example, consider a `Block` statement B, which contains a sequence of other statements A_i for $1 \leq i \leq n$ for some n. Let statement A_1 transition between states s and s_1 if and only if $[s, s_1]_{A_1}$. The set of states that the sequence $A_1; A_2$ can transition to from s is equal to the union of all states that A_2 can transition to from

any state s_1 such that $[s, s_1]_{A_1}$. Therefore, $\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_{A_1;A_2} \Leftrightarrow (\exists s_1 \in \overline{DS} : [s, s_1]_{A_1} \wedge [s_1, s']_{A_2})$. If we extrapolate this reasoning to the whole block B:

$$\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_B \Leftrightarrow (\exists s_1, s_2 \ldots s_{n-1} \in \overline{DS} \times \cdots \times \overline{DS} : [s, s_1]_{A_1} \wedge [s_1, s_2]_{A_2} \wedge \cdots \wedge [s_{n-1}, s']_{A_n})$$

Loop translation A `ForEach` loop statement (FE) is defined with three parameters: the set of objects being iterated over, the variable containing the iterated value, and the block of code that will be executed for each object in the object set. Let α be the object set, v the variable, and B the block of code. Let $|\alpha| = n$. By definition, the order of iteration is non-deterministic.

Since B has access to the iterated object that is different for each iteration, executions of B are affected by the iterated variable. Effectively, each iteration is a different state transition: we use notation $[s, s']_{B_o}$ to refer to a possible execution of an iteration executed for object o. In this case, we refer to o as the *trigger object*. The formula defining the FE loop is:

$$\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_{FE} \Leftrightarrow \exists o_1 \ldots o_n \in \alpha, \exists s_1 \ldots s_n \in \overline{DS} :$$
$$\forall i, j \in [1 \ldots n] : i \neq j \Leftrightarrow o_i \neq o_j \wedge \quad (1)$$
$$[s, s_1]_{B_{o1}} \wedge [s_1, s_2]_{B_{o2}} \wedge \cdots \wedge [s_{n-1}, s_n]_{B_{on}} \wedge s_n = s' \quad (2)$$

In words, a pair of states is an execution of a given loop FE if and only if there exists an enumeration of objects from α and a sequence of states such that (1) the said enumeration of objects is a permutation of α, and (2) the said sequence of states is achievable by triggering iterations in the order of the object permutation.

There exists a corner case where an object that is about to trigger an iteration gets deleted by a prior iteration. We did not include this corner case as part of the definition as it introduces considerable complexity, but the semantic is as follows: such an iteration will still execute with an empty set iterator variable value. This behavior is in concordance with our abstraction and the behavior of ORM tools when objects are deleted before triggering iterations.

3.1.6 Invariant translation

Invariants are translated as boolean expressions. Unlike boolean expressions, invariants can be translated twice: once in the pre-state of the action and, for the purpose of data integrity verification, once in the post-state of the action. In the pre-state of an action, we state that the conjunction of all invariants holds, defining that the pre-state is consistent. We translate the invariant once again in the post-state if we are verifying data integrity.

4 Verification via many-sorted logic

We implemented the FOL translation we described in the previous section and used it to verify data model properties of web applications. We observed that, the theorem prover is unable to give a conclusive result for some verification queries. Since FOL is undecidable in general, FOL theorem provers are not guaranteed to produce a result for all cases. Our experiments indicate that, when we use the FOL translation described in the previous section, about 17% of verification queries lead to inconclusive results.

Minimizing the ratio of inconclusive results is a necessary step for making our approach usable in practice. Inconclusive results force the developer to manually investigate actions and invariants, and since we encounter inconclusive results in the most complex actions, this is a difficult and error prone process.

In order to understand the cause of inconclusive results, we investigated the logs of the FOL theorem prover we used in our experiments. We noticed that the theorem prover did an excessive number of deductions solely to reason about the types of quantified variables and objects. Since FOL does not have a notion of type, our FOL translation generates predicates that encode all the type information, and the theorem prover was spending a lot of time reasoning about these predicates.

This seemed unnecessary to us, as in general, inheritance is rarely used in web applications. Out of 25 most starred Ruby on Rails applications on Github only 7 employ inheritance, and on average, only 23% of classes inherit or are inherited from other classes. This means that, if FOL would allow us, we could annotate our formulas with precise type information and a theorem prover might use this information to greatly trim the space of deductions it makes.

There exists a variant of FOL called many-sorted logic. Many-sorted logic enforces a rigid type system on top of FOL, where all predicates, functions etc. have to be annotated with types.

In this section we present a translation of data models to many-sorted logic, and encounter and fix a problem regarding empty logic in our many-sorted translation. Then, in our experimental evaluation, we show that using many-sorted logic drastically increases our verification performance, and furthermore, that sorts themselves are the main factor in this performance increase.

4.1 Many-sorted logic

Sometimes it is useful to divide the universe of a structure using types with mutually exclusive domains. This is especially true if the functions and predicates make sense only within a specific domain. Types in many-sorted logic are called *sorts*. Many-sorted logic requires us to explicitly declare the types of all function and predicate arguments, function return values and variables. It also gives us the ability to quantify over elements of a given type instead of over the whole universe.

Formally, many-sorted logic is very similar to classical FOL. In addition to everything discussed in Sect. 3 for a first order logic language L, the many-sorted logic language L also includes a set of sorts S. Functions and predicates in F and P respec-

tively define the sorts of their arguments, functions define the sort of their return value, and all variables are associated with a sort from S. We also require all formulas to be well typed (e.g. a predicate can only accept a term as an argument if the term's sort matches the predicate's declaration).

A structure S in many-sorted logic does not contain a single universe U. Instead, it contains a non-empty universe U^s for each sort $s \in S$. For each predicate p of sorts $s_1 \ldots s_k$ and arity k, we define P^S as a subset of $U^{s_1} \times \cdots \times U^{S_k}$. The set F^U is defined analogously, and V^U assigns an element of a variable's sort to each variable. Quantification is always done over a specific sort's universe. For clarity, we explicitly declare the sort s of a variable v when quantifying by using the notation $\forall s\, v\colon f$.

Note that many-sorted logic and unsorted logic have equivalent expressive power (Claessen et al. 2011). Given a set of many-sorted formulas, a similar set of unsorted formulas is equisatisfiable if we introduce predicates used to denote sorts and conjoin the formulas that partition the universe to these sorts. Unsorted logic can be translated to many-sorted logic by introducing a single sort that applies to all language elements.

4.2 Empty logic

Empty universes are a useful concept for data model verification. In general, a data model state may contain no objects. This is an important consideration for data model verification (e.g. does the application behave properly even if there exist no Users or Accounts?). For this reason it is necessary to consider empty universes as a possibility during verification. As one would expect, data model verification tools, such as Alloy (Jackson 2002), support empty domains. However, empty universes are outside the scope of classical FOL. Even though Spass (one of the FOL theorem provers we use in our verification framework) does not support empty universes, our translation was such that the empty model state was a possibility. This will cease to be the case for the translation we describe below in Sect. 4.3 for many-sorted logic. However, before we explain this problem, we must define *empty logic*: FOL that allows an empty universe.

FOL universes are typically defined to be non-empty. Allowing the special case of an empty universe makes definitions more complicated, and invalidates certain inference rules that stop working only in the case of an empty universe (for example, $\phi \vee \exists x \psi$ implies $\exists x(\phi \vee \psi)$ where x is not a free variable in ϕ). The treatment of variables and function return values becomes problematic because terms are expected to always take a value of one element of the universe. This is not possible in empty universes.

Furthermore, the possibility of an empty universes breaks certain fundamental rules about FOL. E.g. $\forall x\colon x \neq x$ is normally an unsatisfiable formula. If we define quantification over an empty universe to be vacuously true (as there does not exist an assignment of the variable that does not satisfy the subformula), this example formula is satisfied by a structure with an empty universe.

Empty logic is a variant of FOL that allows empty universes. The treatment of empty universe in empty logic is defined by Quine (1954): universal quantification over an empty set is considered vacuously true (since there exists no counterexample

variable assignment), and existential quantification over an empty set is considered vacuously false (since there exists no satisfactory variable assignment).

This interpretation of quantification over empty sorts is in concordance with an alternative definition of universal quantification: Given a universe U, quantification $\forall v\colon f$ can be unrolled into a conjunction of all formulas that result from replacing v in f with an element of U. In case of an empty universe this list of quantified formulas is empty, and the neutral element of conjunction is the boolean $true$.

In combination with many-sorted logic, empty logic allows a sort's universe to be empty. Although theorem provers we use during verification do not support empty logic, in our translation of data models to FOL, we simulate the empty logic semantics so that the resulting translation covers the data model behaviors where data classes can be empty (i.e., without any instances). We discuss our formalization of the data models and how we deal with many-sorted logic and empty universes in our translation to FOL in the following sections.

4.3 Translation of abstract data stores to many-sorted logic

The translation presented in Sect. 3 is based on unsorted, empty logic. In this section we modify the previously presented translation to many-sorted logic. For brevity, we focus only on classes as associations are largely analogous.

Within our translation where universe elements correspond to entities, sorts naturally serve the purpose similar to classes and associations. However, sorts imply disjoint universes, which is only suitable for classes that do not employ inheritance. Classes that employ inheritance cannot be directly mapped to sorts because a subclass's object set is a subset of a parent's.

To work around this problem, we partition the set of all classes into *inheritance clusters*. An inheritance cluster is a maximal set of classes such that, for any two classes c_1 and c_k in the cluster, there exists a list of classes $c_1, c_2, \ldots c_k$ where each consecutive pair of classes constitutes a child-parent or parent-child relationship. In other words, in the class graph where vertices are classes and edges correspond to inheritance, an inheritance cluster is a maximally connected component. Note that all classes that do not employ inheritance are members of singleton clusters.

For each inheritance cluster we introduce a sort that is common to all classes in the cluster. In case of an inheritance cluster with multiple classes we introduce predicates and axioms in order to differentiate classes within the cluster. These predicates and axioms are similar in purpose to the predicates used in the unsorted logic translation. For each class c in a non-singleton inheritance cluster we introduce unary predicates $\overline{c}$ and $\overline{c}_x$ of the cluster's sort and introduce axioms that resemble the ones defined for unsorted logic, the key distinction being these axioms refer to classes of that cluster only.

Specifically, inheritance axioms are defined as follows: for each class c that belongs to an inheritance cluster of sort s and whose superclass set is $\text{superclass}(c) = \{p_1, p_2 \ldots p_k\}$:

$$\forall s\, o\colon \overline{c}(o) \rightarrow \overline{p_1}(o) \wedge \overline{p_2}(o) \wedge \cdots \wedge \overline{p_k}(o)$$

Sorts: Cluster, Tag, Comment.
Predicates: Account(Cluster), Contact(Cluster), Taggable(Cluster), XTaggable(Cluster), Commentable(Cluster), XCommentable(Cluster).

$$\forall\ \texttt{Cluster}\ o\colon \texttt{Account}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o) \quad (1)$$

$$\forall\ \texttt{Cluster}\ o\colon \texttt{Contact}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o) \quad (2)$$

$$\forall\ \texttt{Cluster}\ o\colon \texttt{XTaggable}(o) \leftrightarrow \texttt{Taggable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \quad (3)$$

$$\forall\ \texttt{Cluster}\ o\colon \texttt{XCommentable}(o) \leftrightarrow \texttt{Commentable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \quad (4)$$

$$\forall\ \texttt{Cluster}\ o\colon \texttt{Account}(o) \rightarrow \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \quad (5)$$

$$\forall\ \texttt{Cluster}\ o\colon \texttt{Contact}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \quad (6)$$

$$\forall\ \texttt{Cluster}\ o\colon \texttt{XTaggable}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XCommentable}(o) \quad (7)$$

$$\forall\ \texttt{Cluster}\ o\colon \texttt{XCommentable}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \quad (8)$$

Fig. 6 Axioms defining the class diagram in Fig. 4 in many-sorted logic

For the model presented in Fig. 4, inheritance axioms are formulas (1) and (2) in Fig. 6.

An instance axiom is generated for each class c. Let $\{s_1 \ldots s_k\}$ be the set of c's subclasses and let s be the sort of c's inheritance cluster:

$$\forall s\ o\colon \overline{c}_x(o) \leftrightarrow \overline{c}(o) \wedge \neg\overline{s_1}(o) \wedge \cdots \wedge \neg\overline{s_k}(o)$$

Given the model presented in Fig. 4, instance axioms are formulas (3) and (4) in Fig. 6.

Finally, membership axioms are generated for each non-singleton inheritance cluster individually instead of for the entire set C. Given an inheritance cluster that consists of classes $\{c_1, \ldots c_k\}$ where $k > 1$ we generate an axiom for each class c_i inside this cluster:

$$\forall s\ o\colon \overline{c_i}(o) \rightarrow \neg\overline{c_1}(o) \wedge \cdots \wedge \neg\overline{c_{i-1}}(o) \wedge \neg\overline{c_{i+1}}(o) \wedge \cdots \wedge \neg\overline{c_k}(o)$$

Formulas (5)–(8) in Fig. 6 correspond to membership axioms for the model in Fig. 4.

The number of introduced predicates and axioms is highly dependent on the data model in question. With no inheritance, no additional predicates and axioms are introduced. The number and size of formulas introduced by each inheritance cluster are linear in the number of classes in the cluster. However, most classes do not employ inheritance in data models of real world applications (18 out of 25 most starred Ruby on Rails applications do not employ inheritance at all, with an average of 23% classes involving inheritance), making most classes part of singleton inheritance clusters. Furthermore, if multiple non-singleton inheritance clusters exist in the data model, the size of generated axioms is relatively small when compared to those generated by the unsorted logic translation. Finally, in case of a model with only singleton clusters, no additional axioms are required to define the type system.

4.3.1 Empty logic and empty structures

Our treatment of empty structures is dependent on whether the underlying theory is unsorted or many-sorted. In fact, our translation to unsorted logic as presented in

Fig. 7 Example action based on FatFreeCRM (2013)

```
1  class CommentsController
2    ...
3    def destroy
4      @comment = Comment.find(params[:id])
5      @comment.destroy
6      respond_with(@comment)
7    end
8    ...
9  end
```

Predicates: `PreState`, `PostState`, `AtComment`.

$$\forall x\colon \texttt{AtComment}(x) \Rightarrow \texttt{Comment}(x) \tag{1}$$

$$\forall x\colon (\forall y\colon \texttt{AtComment}(x) \wedge \texttt{AtComment}(y) \Rightarrow x = y) \tag{2}$$

$$\forall x\colon \texttt{AtComment}(x) \Rightarrow \neg\texttt{PostState}(x) \tag{3}$$

$$\forall x\colon \neg\texttt{AtComment}(x) \Rightarrow (\texttt{PreState}(x) \Leftrightarrow \texttt{PostState}(x)) \tag{4}$$

Fig. 8 Unsorted action translation example

Sect. 3 allows empty structures by default. This becomes clear when we change the interpretation of all type predicates $\overline{c}$ to imply that the universe element in question is of the given type, but in addition, it *exists* semantically. Notice that our encoding does not require that all universe elements are of a class type. For example, we use universe elements to represent tuples, and it is not required for a universe element to represent either an object or a tuple.

Whenever we define functions and predicates in unsorted logic we constrain argument values and the return value, if applicable, to be of expected types. As a corollary of our expanded interpretation, function return values objects exist semantically if and only if arguments exist semantically and are of corresponding types. Similarly, predicates may accept a set of domain elements under the condition that they exist semantically and are of corresponding types.

As for quantification, whenever quantifying over a class type, we introduce a condition that the subformula is relevant only for domain elements that represent objects of the given type. For example, in order to universally quantify over elements of class c using the variable v and a subformula f we generate a formula $\forall v\colon \overline{c}(v) \rightarrow f$. In case of existential quantification we would instead generate $\exists v\colon \overline{c}(v) \wedge f$.

For example, the action presented in Fig. 7 can be translated to FOL as defined in Fig. 8. For brevity, we omit listing all predicates and axioms that define the type system. In this translation, the `AtComment` predicate denotes values that are saved in the `@Comment` variable. First we constrain type-specific predicates to refer to their actual types (formula (1)). Note that as part of our interpretation of class type predicates, any entity accepted by the `AtComment` is also accepted by `Comment` and therefore exists semantically. Next, in formula (2) we constrain that there exists at most one element in variable `AtComment`, as the `find` method in Ruby on Rails (line 4 in Fig. 4) returns at most one object.

Formulas (3) and (4) define the delete statement. Formula (3) defines that the objects in the `@Comment` variable no longer exist after the statement (regardless of their existence before). Formula (4) defines that all objects outside this variable existed before

Predicates: $\texttt{PreState}_{\texttt{Cluster}}(\texttt{Cluster})$, $\texttt{PreState}_{\texttt{Comment}}(\texttt{Comment})$, $\texttt{PreState}_{\texttt{Tag}}(\texttt{Tag})$, $\texttt{PostState}_{\texttt{Cluster}}(\texttt{Cluster})$, $\texttt{PostState}_{\texttt{Comment}}(\texttt{Comment})$, $\texttt{PostState}_{\texttt{Tag}}(\texttt{Tag})$, $\texttt{AtComment}(\texttt{Comment})$, $\texttt{Comment}_P(\texttt{Comment})$, $\texttt{Tag}_P(\texttt{Tag})$.

$$\forall\ \texttt{Comment}\ x\colon \texttt{AtComment}(x) \Rightarrow \texttt{Comment}_P(x) \tag{1}$$

$$\forall\ \texttt{Comment}\ x\colon (\forall \texttt{Comment} y\colon \texttt{AtComment}(x) \wedge \texttt{AtComment}(y) \Rightarrow x = y) \tag{2}$$

$$\forall\ \texttt{Comment}\ x\colon \texttt{AtComment}(x) \Rightarrow \texttt{PreState}_{\texttt{Comment}}(x) \wedge \neg \texttt{PostState}_{\texttt{Comment}}(x) \tag{3}$$

$$\forall\ \texttt{Comment}\ x\colon \neg\texttt{AtComment}(x) \Rightarrow (\texttt{PreState}_{\texttt{Cluster}}(x) \Leftrightarrow \texttt{PostState}_{\texttt{Cluster}}(x)) \tag{4}$$

Fig. 9 Many-sorted action translation example

if and only if they exist after the statement has finished executing. This particular translation allows for an empty universe. Such a structure would have no elements accepted by predicates `Comment` and `AtComment`.

The problem with the empty universe becomes more apparent with the many-sorted logic translation. If we were to define a Comment sort and use it alone to define the set of all comments, then the universe of this sort would be non-empty, meaning that at least one Comment would exist for every sort. To go around this problem, for each such class c, we introduce a predicate $\overline{c}$ that accepts a single argument of c's sort. We do not introduce any axioms. We use these predicates to define object sets of these classes, implying that object sets are subsets of their corresponding universes.

Given the example action in Fig. 7, a many-sorted translation can be defined as in Fig. 9. Note that, once again, we omit declaring all sorts, predicates and axioms from Fig. 6 for brevity.

Notice that we introduce predicates $\texttt{Comment}_P$ and $\texttt{Tag}_P$ in addition to previously defined sorts `Comment` and `Tag`. In Formula (1) we define that all elements accepted by `AtComment` are also accepted by $\texttt{Comment}_P$. This is necessary to express since, without this axiom, there could be an element accepted by `AtComment` that is not accepted by $\texttt{Comment}_P$. Formula (2) defines that there exists at most one element accepted by `AtComment`. Formulas (3) and (4) define how the delete statement transitions between the pre-state and the post-state. These formulas are analogous to formulas (3) and (4) in the unsorted translation. Note that, however, these formulas are constrained to the `Comment` sort. All other sorts are handled implicitly (we do not differentiate between their pre- and post-states). This demonstrates the benefit of introducing sorts, as the theorem prover does not need to reason at all about other types by default.

Empty structures are handled by this translation. For example, a structure that represents this case would have no entities of sort `Comment` be accepted by predicates $\texttt{Comment}_P$ and `AtComment`. Without introducing a predicate $\texttt{Comment}_P$ this would not be the case.

5 Experimental evaluation

In this section we present our experimental evaluation benchmark and our experimental results. We first describe two sets of experiments that comparatively explore the advantages and disadvantages of different provers and logics. We then describe our overall combined experimental results.

Table 1 Applications in our experimental evaluation benchmark

Application	LoC (Ruby)	Classes	Actions	Invariants
Avare	1137	6	26	3
Communautaire	753	5	28	6
Copycopter	3201	6	11	6
CoRM	7745	39	163	32
FatFreeCRM	20,178	32	120	8
Fulcrum	3066	5	40	6
Kandan	1535	5	25	6
Lobsters	5501	17	86	9
Obtvse2	828	2	13	1
Quant	4124	9	38	4
Redmine	84,770	74	264	21
S2L	1334	9	44	4
Sprintapp	3042	15	120	8
Squash	15,801	19	46	18
Tracks	17,562	11	117	9
Trado	10,083	33	66	10
WM-app	2425	18	95	4
Totals	183,085	305	1302	155

5.1 Experimental evaluation benchmark

We analyzed a total of 17 open-source Rails applications. We obtained these applications from various sources: we looked at the 25 most-starred open-source Rails applications on GitHub according to the OpenSourceRails.com website (Open Source Rails 2016), at a compilation of open source Rails applications categorized by domain Rails (Karaca 2016), and at applications investigated by related work.

We consider that these applications are representative of real-world Rails applications for several reasons. They vary in size and complexity, their domain of purpose, the number of developers who developed and maintained them, as well as the technologies that they utilize.

Table 1 shows the list of applications included in our benchmark. Column *LoC (Ruby)* shows the number of Ruby lines of code in these applications; this number does not include JavaScript, HTML, dynamic HTML generation through irb files, or configuration files. Columns *Classes*, *Actions* and *Invariants* show the number of model classes, actions, and invariants, respectively. As invariants are not part of the core Rails framework, we wrote them manually for each application after investigating their source code.

5.2 Examples of detected bugs

Before presenting the experimental evaluation of our techniques, we discuss five example bugs that we found using them (Bocic and Bultan 2015b).

FatFreeCRM is the customer relation management application that we described in Sect. 3.1. One of the bugs that our techniques found in FatFreeCRM is caused by `Todo` objects, normally associated with a specific `User`, not being deleted when their `User` is deleted. We call these `Todo` objects *orphaned*. Orphaned `Todo` objects are invalid because the application assumes that their owner exists, causing crashes whenever an orphaned `Todo`'s owner is accessed. Because of its severity, this bug was acknowledged and repaired by the FatFreeCRM developers immediately after we submitted a bug report.

Another FatFreeCRM bug that we found relates to `Permission` objects. `Permission` objects define access permissions for either a `User` or a `Group` to a given `Asset`. Our tool found that it is possible to have a `Permission` without any associated `User` or `Group` objects. This bug can be replicated by deleting a `Group` that has associated `Permissions`. Although the cause of this bug is similar to that of the previously described one, its repercussions are very different. If there exists an `Asset` object none of whose `Permission` objects have associated `Users` or `Groups`, it is possible to expose these `Assets` to the public without any user receiving an error message, and without any `User` or `Group` owning and managing this `Asset`.

Tracks is the task and to-do management application that we described in Sect. 2.1. We now describe three bugs that we found in Tracks using the techniques described in this paper.

The first bug is related to the possibility of orphaning an instance of a `Dependent` class. In the case of this bug, the orphaned objects cannot be accessed by actions in any way. However, it creates a memory leak that can affect performance by unnecessarily populating database tables and indices.

Another bug arises when a `User` is deleted, all `Projects` of the `User` are deleted as well, but `Notes` of deleted `Projects` remain orphaned. Similarly to the previous bug, these orphaned `Notes` are not accessible in any way, but the orphaned objects take up space in the database and inflate indices.

Finally, we also found a bug that arose when our techniques reported an inconclusive result within the action used to create `Dependent` instances between two given `Todos`. Semantically, there must not be dependency cycles between `Todos`; this is a structural property of the application. Our method could not prove or disprove that cycles between `Todos` cannot be created. Upon manual inspection we found that, while the UI prevents this, HTTP requests can be made to create a cycle between `Todos`. The repercussions of this bug are potentially enormous. Whenever the application traverses the predecessor list of a `Todo` inside a dependency cycle it will get stuck in an infinite loop, eventually crashing the thread and posting an error to the `User`. No error is shown when the user creates this cycle, only later upon accessing it. This creates a situation where repairing the state of the data may be impossible.

We reported the three bugs described above to the developers of Tracks, and they have fixed them.

5.3 Comparative evaluation

We conducted two sets of experiments. Both of them involved the verification of applications shown in Table 1. In total, we had 17,291 data integrity properties to be verified using theorem proving. We refer to these properties as verification cases or verification instances. We translated these 17,291 cases into different FOL variants in order to evaluate the performance using different provers, heuristics and translations: a total of 69,164 FOL theorems. For each of these cases, we executed the verification with a time limit of 5 min. If the theorem prover did not deduce a result within 5 min we treated the result as inconclusive. Given that most verification cases terminate in a few seconds, we believe that this is a reasonable time limit.

5.3.1 FOL theorem provers

In these experiments, we used Spass (Weidenbach et al. 2009) as our unsorted theorem prover. Spass is a FOL theorem prover based on superposition calculus. While Spass supports multiple input formats, we translated the verification cases to Spass's own input format (Weidenbach). Spass tries to prove that a conjecture follows from a set of axioms by negating the conjecture and attempting to deduce a contradiction. If this contradiction is found, then the conjecture is proven to follow from the axioms.

Note that Spass supports *soft sorts* (Weidenbach et al. 2009) which are different than the sorts in many-sorted logic we discussed earlier, and any other sort system we encountered. Soft sorts do not imply mutually exclusive universes. In a soft sort system any universe element may be of a sort, of no sort, or of multiple sorts. Semantically, these sorts are indistinguishable from unary predicates. Furthermore, Spass by default infers soft sorts even if none are explicitly specified. Spass provides a command option that allows us to disable the soft sort system, in which case the theorem prover treats soft sorts as unary predicates. The differences between these soft sorts and sorts as defined in many-sorted logic have been observed before (Blanchette et al. 2012). In the following discussion, whenever we refer to sorts we refer to sorts defined by many-sorted logic. We will use "soft sorts" to refer to Spass's version of sorts specifically.

We used Z3 (de Moura and Bjørner 2008) to evaluate effectiveness of data model verification using many-sorted logic. Z3 is a DPLL(T) (Dutertre and de Moura 2006) based SMT solver that deals with free quantification and uninterpreted functions using E-matching (de Moura and Bjrner 2007).

SMT solvers tend to support many different theories, such as arithmetic, arrays or bit arrays. These theories are combined in propositional logic, which serves to tie the underlying theories without interpreting them. Instead, predicates in underlying theories are treated as propositional variables, and left to the underlying provers to be solved. Partial conclusions from these underlying theories may be propagated to other underlying provers in DPLL(T) in order to reach other conclusions. When used only

Table 2 Verification performance summary

Method	Number of timeouts		Verif. time (s)		Unit propagations		Memory (Mb)		
			Avg	Median	Avg	Median	Avg	Median	Max
Spass (soft sorts on)	2974	(17.20%)	10.05	9.22	n/a	n/a	60.97	61.56	86.80
Spass (soft sorts off)	2707	(15.66%)	12.11	9.43	n/a	n/a	60.87	61.56	111.83
Z3 (many-sorted)	23	(0.13%)	0.06	0.04	380.04	23	4.02	3.87	285.64
Z3 (unsorted)	524	(3.03%)	2.37	0.48	1125.32	84	140.19	46.49	15,490.26

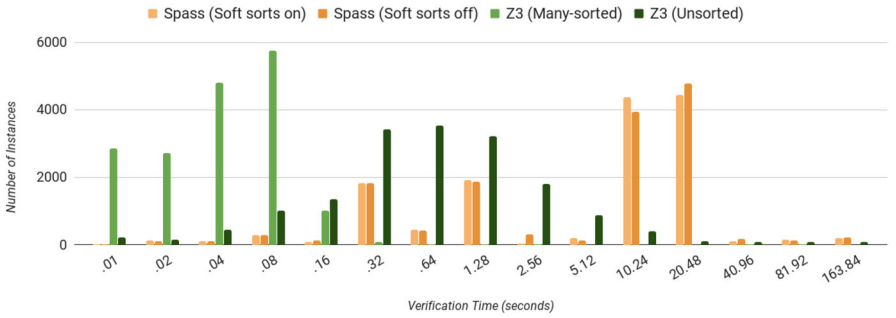

Fig. 10 Verification time distribution

with free quantification, free sorts and uninterpreted functions (which is denoted as the problem group UF), SMT solvers behave like many-sorted logic theorem provers.

SMT solvers try to find instances that satisfy the specification, so in order to prove that the conjecture follows from axioms, we negate our conjecture and state it as an additional axiom. The conjecture follows from the axioms if and only if this resulting set of axioms is unsatisfiable.

5.3.2 Spass versus Z3 performance

Our first set of experiments compare the performance of Spass and Z3 for the purpose of data model verification. These experiments were conducted solely to detect whether Z3 can sometimes outperform Spass, either by reaching results that Spass could not, or reaching them in less time. If so, our efforts in translating data models to SMT would increase the performance and/or reduce the ratio of inconclusive results in our data model verification efforts, and therefore increase the viability of data model verification in the real world.

Our results are summarized in Table 2 and Fig. 10. The performance difference was beyond our initial expectations. Note that the Z3 (Unsorted) entries are only relevant for the experiment discussed in the next subsection and can be disregarded for now, as is the case for *Unit Propagations* columns. With soft sorts enabled, Spass produced 2974 inconclusive results (17.20%). With soft sorts disabled, Spass produced 2707

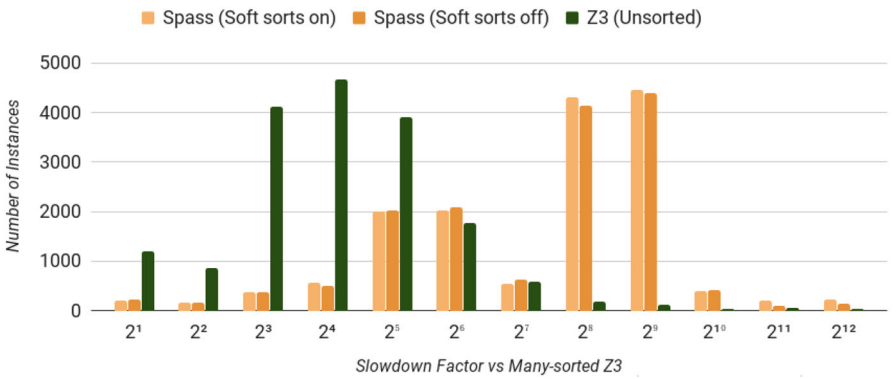

Fig. 11 Distribution of the slowdown factor compared to (many-sorted) Z3

inconclusive results (15.66%). Interestingly, there are 45 cases where enabling sorts led Spass to a conclusive result where disabling sorts did not, yet there are 309 cases where the opposite is true. Performance-wise, Spass performed similarly regardless of the soft sorts setting. For both settings, excluding timeouts, verification took an average of about 10 s per case. The median case is just over 9 s. Memory consumption averaged at around 60 Mb, with the median case of 61.56 Mb. Memory consumption peaked at just over 100 Mb memory when Spass produced a conclusive result. For inconclusive results, memory consumption peaked at just over 1 Gb.

Z3 performed far better than Spass with either heuristic. Z3 produced far fewer inconclusive results, only 23 (0.13%). In addition, in only 3 cases did Z3 fail to produce a result when Spass succeeded. In the remaining 20 cases, neither prover could reach a conclusive result in 300 s. On average, Z3 took 0.06 s per verification case, with a median time of 0.04 s. Spass outperformed Z3 in only 3 cases in terms of time performance, while Z3 outperformed both Spass heuristics in 14,260 cases, counting only cases where all provers produced a result. Furthermore, Z3's average memory consumption was just over 4 Mb, with a median under 4 Mb. Memory consumption peaked at just under 300 Mb. However, Z3 tends to consume far more memory when it is failing to produce a conclusive result. In one case, Z3 used 35 Gb of memory before forcefully being terminated after 5 min.

Figure 10 shows the distribution of the verification cases over the verification time ranges for each theorem prover. For example, the leftmost column (labeled .01) shows that Z3 produced a verification result in less or equal than 0.01 s 2844 times. Spass achieved a result within this time only 8 times, which is not visible on the chart. The next time range is labeled .02 and shows that Z3 produced a verification result in more than 0.01 s but less or equal to 0.02 s 2712 times, while Spass with soft sorts on produced a result 131 times within the same timeframe.

We wanted to compare the performance of different provers on case-by-case basis. For each verification case, we calculated the *relative slowdown factor* induced by a prover compared with Z3. So, for example, if a verification case was verified 85 times slower using Spass with sorts enabled when compared with Z3, this counts as a slowdown factor of 85. Figure 11 and Table 3 summarize this data.

Table 3 Observed slowdowns with respect to (many-sorted) Z3

Method	Average	Median	Interdecile range
Spass (soft sorts on)	288.84	195.75	20.0–350.6
Spass (soft sorts off)	311.81	200.50	20.0–358.2
Unsorted Z3	82.38	12.50	4.9–48.2

Figure 11 contains the distribution of slowdown factors per prover. For example, Spass (with and without soft sorts) is most frequently between 2^8 and 2^9 times slower than Z3. Table 3 contains additional information about this slowdown. On average, Z3 was 288.84 times faster than Spass with soft sorts on, and 311.81 times faster with soft sorts off. In the median case, Z3 was 197.75 and 200.5 times faster, respectively.

In order to estimate a range of performance increase factor for the majority of cases, we calculated interdecile ranges of these distributions. The interdecile range of a sample is the range of values ignoring the lowest and highest 10% of the sample. It serves to communicate a range of values, ignoring outliers. The interdecile ranges of performance increases of Z3 over Spass with soft sorts on and off are 20.0–350.6 and 20.0–358.2, respectively. This means that, 80% of the time, Spass was one to two orders of magnitude slower than Z3.

In summary, our translation to SMT and use of Z3 for verification increased the performance of verification of our method by two orders of magnitude, and brought the number of inconclusive results down from around 16 to 0.13%.

5.3.3 Many-sorted versus unsorted performance

We observed a drastic improvement in our method's performance by utilizing Z3 instead of Spass. However, this difference was beyond our expectations, and we wanted to investigate the reason behind the performance difference. This is hard to pinpoint since Spass and Z3 are fundamentally different. They utilize a different approach to theorem proving and have different optimizations and heuristics.

During manual investigation of Spass's deduction logs we noticed that Spass was taking a significant amount of time reasoning about types of quantified variables. This is true regardless of whether soft sorts are enabled or not. This reasoning about types would not be necessary or would be drastically reduced if the theorem prover supported (non-soft) sorts. Even if the model contains a larger number of classes that inherit from one another, causing us to introduce predicates and axioms that resemble the ones generated for unsorted logic, this type reasoning is constrained to a smaller scope of an inheritance cluster instead of the set of all classes.

We implemented an unsorted translation to SMT in order to observe the benefit of using sorts. Because SMT-LIB requires all predicates and functions to be sorted, we defined a single sort (called `Sort`) that we used for all language elements. Since this single sort represents everything, we effectively provide no explicit type information. On top of this sort(less) system we enforce the type system using predicates and axioms using the unsorted translation presented in Sect. 4.1. Thereby we specify the

type system in a way that requires type reasoning in a way that corresponds to the amount of information we provide to Spass.

We ran the same suite of application models and action-invariant pairs using the many-sorted and unsorted translations to SMT. Table 2 summarizes the performance of many-sorted and unsorted Z3 verification. Unsorted Z3 did not produce a conclusive result in 524 cases (3.03%). On average, many-sorted Z3 took 0.06 s per case whereas unsorted Z3 took 2.37 s. Median values are 0.04 for the many-sorted logic and 0.48 for the unsorted translation.

The *Unit Propagations* columns in Table 2 refer to the number of DPLL(T) unit propagations done by Z3. The number of unit propagations required by the many-sorted translation before reaching a conclusive result was significantly lower than that of the unsorted translation. For the many-sorted translation, the mean number of propagations was 380.04, and the median was 23. For the unsorted translation, the mean number of propagations was 1125.32, and the median was 84. Therefore, Z3 needed to do more work to reach conclusive results when using unsorted logic.

Finally, the memory footprint of verification suffered as well. The many-sorted translation used an average of 4.02 Mb of memory per verification case, with a median of 3.87 Mb. The unsorted translation was drastically more demanding, with an average of 140.19 Mb and a median of 46.49 Mb. Memory consumption peaked at over 15 Gb of memory for unsorted theorem proving.

Figure 10 contains data for the unsorted Z3 translation in addition to (many-sorted) Z3 and Spass results. Similarly, Fig. 11 and Table 3 show the distribution of case-by-case slowdown factors when comparing unsorted Z3 to many-sorted Z3. On average, the many-sorted translation resulted in 80.43 times faster verification compared to the unsorted translation when both methods produced conclusive results. The median case is 11.8, and the interdecile range is 3.6–47.2.

These results imply a large performance difference between many-sorted and unsorted logic verification in Z3. While this does not imply that implementing proper many-sorted logic in Spass would increase Spass's performance by a similar factor, it does indicate that the reduction of reasoning induced by many-sorted over unsorted logic plays a significant role in the performance gain we observed.

5.4 Overall data model verification results

In Sect. 5.3 we presented a comparative analysis of the results obtained using different provers and logics. We now focus on the overall results of our approach.

To summarize what we explained in Sect. 5.3.1, our tool verifies each extracted ADS specification using Spass (Weidenbach et al. 2009) and Z3 (de Moura and Bjørner 2008) for theorem proving. First we translate action/invariant pairs to FOL. These FOL formulas are sent both to the Z3 SMT solver and to the Spass theorem prover. We express our FOL theorems in SMT using problem group UF, which includes free quantification, free sorts and uninterpreted functions. Z3 checks satisfiability, so when we construct a formula to be sent to Z3, if a satisfying model exists for the formula, then there exists an execution of the action violates data integrity. If, on the other hand, Z3 reports that the formula is unsatisfiable, then we can conclude that the action correctly

enforces data integrity. Spass, on the other hand, checks whether a conjecture implies from a set of axioms. This conjecture is a formula that asserts that data integrity is preserved. If Spass reports that the conjecture follows from the axioms, then we can conclude that no execution that violates data integrity exists. However, if Spass reports that the conjecture does not always follow from the axioms, then we can conclude that an execution that violates data integrity exists.

We generate formulas without restrictions on quantification nesting, without a bound on the number of arguments for predicates, and without a bound on the domains. The formulas we generate are not in a decidable fragment of FOL that we know of. This implies that Z3 and Spass may not be able to produce a conclusive result for some of the formulas we generate.

We used Z3 and Spass concurrently, waiting for either theorem prover to produce a result, after which the other prover is terminated. In our experiments we observed that Z3 is faster and is more likely to report conclusive results for the formulas we generate.

Table 4 shows the verification results that we obtained. These experiments were run on a computer with an Intel Core i7-6850K processor, with 128 GB RAM, running 64 bit Linux. We run a total of 12 processes concurrently to cover the experimental set.

Column *Verified Properties* shows the number of action/invariant pairs and authorization properties generated from the application. Each of these properties is translated to FOL and verified independently. Column *Average Predicates* shows the average number of predicates in FOL formulas that we generated. Corm and Redmine have the highest number because their schemas are the most complicated, increasing the number of predicates and axioms needed to specify them.

Column *Max Memory (Mb)* shows the maximum memory a theorem used to produce a conclusive result. Column *Average Time (sec)* shows the average time it took before a theorem prover (Z3 or Spass) took to deduce a conclusive result. Columns *Verified*, *Falsified*, *Timeouts* and *False Positives* refer to the number of properties that were verified, falsified, timed out, or that reported a bug that we found not to be an actual bug.

From a total of 17,291 properties, we verified 17,133 to be correct. 93 properties failed verification, which we manually traced to actual bugs in the application. 20 properties timed out for both theorem provers, and there were 45 false positives which were manually confirmed not to be actual bugs. These false positives are due to the fact that we are using an abstract data model that does not fully capture the precise semantics of the application.

Finally, column *Bugs* lists the number of distinct bugs we identified based on falsified properties. In some cases, addressing one falsified property will fix other falsified properties too, as the fix might affect more than one falsified property (one for each action).

Our criterion for determining which falsified properties correspond to distinct bugs is based on the fix required to address the falsified property. For example, if multiple falsified properties can be fixed with a single controller-level change, we consider all those falsified properties to correspond to the same bug. As another example, if multiple falsified properties can be fixed at the level of the model class, we consider

Table 4 Summary of results of verification experiments

Application	Properties	Avg predicates	Max mem (Mb)	Avg time (s)	Verified	Falsified	Time outs	False posit	Bugs
Avare	78	58	5	0.01	71	7	0	0	4
Communaut	168	35	3	0.00	162	0	0	6	0
Copycopter	66	47	46	0.01	66	0	0	0	0
Corm	5216	433	35	0.07	5178	8	8	22	7
FatFree	944	191	5	0.02	942	2	0	0	2
Fulcrum	240	40	4	0.01	233	7	0	0	4
Kandan	150	34	4	0.01	147	3	0	0	2
Lobsters	774	201	69	0.77	753	13	4	4	10
Obtvse2	13	10	1	0.00	13	0	0	0	0
Quant	152	45	2	0.01	152	0	0	0	0
Redmine	5565	432	20	0.05	5542	12	5	6	7
S2L	176	123	5	0.01	168	4	0	4	4
Sprintapp	960	95	29	0.02	944	16	0	0	13
Squash	828	175	9	0.03	824	4	0	0	4
Tracks	1053	88	286	0.23	1036	13	3	1	6
Trado	620	260	50	0.09	614	4	0	2	4
WM-app	288	115	5	0.01	288	0	0	0	0
Total	17,291	322.72	286	0.09	17,133	93	20	45	69

all those falsified properties to refer to the same bug. We manually analyzed all the falsified properties based on this criterion, and based on our analysis, we identified 69 bugs that correspond to 93 falsified properties.

6 Related work

This paper builds on and extends the results reported in Bocic and Bultan (2014), Bocic and Bultan (2015c). Furthermore, it uses the model extraction technique presented in Bocic and Bultan (2017) and loop verification technique presented in Bocic and Bultan (2015a).

Verification of software using theorem provers has been explored before in projects such as Boogie (Barnett et al. 2005), Dafny (Leino 2010), JayHorn (Kahsai et al. 2016), and ESC Java (Flanagan et al. 2002). These projects focus on verification of languages such as C, C# and Java, whereas we focus on data model verification. The underlying type systems are largely different, as their work focuses on manipulating basic types and pointers, whereas our model is based on manipulating sets and relations. There is also more focused software verification work based on theorem proving, such as verification of data structure consistency (Kuncak et al. 2006; Zee et al. 2008), concurrent data structures (Ball et al. 2014), or software-defined networks (Lesani et al. 2014). Our approach focuses on MVC frameworks and leverages their inherent modularity to extract an abstract data model that is different than any prior work that we are aware of, and leads to scalable verification via theorem proving.

The Unified Modeling Language (UML) is a language commonly used for specification of object oriented models. The Object Constraint Language (OCL), which is part of the UML standard, enhances UML with the ability to specify invariants and pre- and post-conditions of methods (OMG; Warmer and Kleppe 1998). Research on verification of OCL specifications have ranged from simulation of object oriented models (Richters and Gogolla 2000), to interactive verification with automated theorem prover support (Ahrendt et al. 2005). However, UML combined with OCL does not provide a way to specify method bodies. Hence, because of the semantic gap between the UML/OCL specifications and actual implementations, the method bodies are unlikely to be modeled precisely using UML/OCL, which means that the bugs we found are likely to be missed by a verification approach based on UML/OCL specifications.

Alloy (Jackson 2002, 2006) is a formal language for specifying object oriented data models and their properties. Alloy Analyzer is used to verify properties of Alloy specifications using bounded verification. Since Alloy was designed specifically for data model verification, it supports sorts and single inheritance. However, it does not support multiple inheritance, which would have to be implemented. Furthermore, the Alloy Analyzer uses SAT-based bounded verification techniques as opposed to our FOL based unbounded verification technique.

DynAlloy is an extension of Alloy that supports dynamic behavior (Frias et al. 2005, 2007) by translating dynamic specifications onto Alloy. While the authors of DynAlloy and related papers talk about *actions*, those actions do not correspond to actions in web applications. Instead, they are more similar to individual statements in programming

languages (Galeotti and Frias 2006). Their work has focused on verification of data structures, not behaviors in data models of web applications.

Near and Jackson (2012) 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. Since their approach requires the developer to write tests, it requires more effort than our automated method and may miss bugs.

iDaVer (Nijjar and Bultan 2011, 2012; Nijjar et al. 2013) represents a set of techniques for verification of data model schemas. Among other features, it is able to translate data model schemas into SMT for unbounded verification. Our models focus on behaviors in data models, even they encompass the static data model schema. In addition, our solution supports multiple inheritance. Their solution does not address the problem of sorts and empty universes, making their verification unsound. Finally, their work does not delve into the difference between logics and their implied encodings.

There are previous results on unbounded verification of data-driven web applications based on high-level specifications (Deutsch et al. 2007, 2006; Deutsch and Vianu 2008). 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. Note that this line of work is done at the specification level and does not address verification of actual code. Due to the semantic gap between their high-level specification of input/output rules and the actual implementations of actions, their work is not directly applicable (without combining it with either a code synthesis or specification extraction approach) to verification of actual implementations of web applications. Additionally, these verification techniques impose restrictions on the use of quantification in their properties, whereas ours does not impose any such restrictions.

As part of a research effort to use Spass as the theorem prover engine for interactive theorem proving (Blanchette et al. 2012), Spass was modified to support many-sorted logic. This was done in order to make deduction logs sort aware, which in turn makes it possible to reconstruct readable proofs from these logs and show them to the user for the purpose of interactive theorem proving. They observed an increase in the number of theories Spass could solve. However, this modification was done for performance reasons, making it reasonable to expect an even larger performance gain from sorts in Spass. The source of this Spass modification is not available, and so we could not include it as part of our experiments.

There are other theorem provers that can be used for data model verification. Vampire (Kovács and Voronkov 2013) is a high performance FOL theorem prover that supports sorts. Snark (Stickel et al. 1994) is another FOL theorem prover, also supporting sorts. We plan to, as part of our future work, implement automatic translation of data models into TPTP syntax (Sutcliffe et al. 1994; TPTP Syntax 2015), the syntax of the test suite that is used by the annual World Championship for Automated Theorem Proving (Pelletier et al. 2002; Sutcliffe and Suttner 2006). This language is readable by many theorem provers, including Spass and Z3. However, given that many-sorted logic has only recently been added to TPTP (Sutcliffe et al. 2012), we expect that the highest performing theorem provers are optimized for unsorted logic.

Unless the theorem prover integrates sorts within its resolution engine, we can expect many-sorted logic to perform no better than unsorted logic. Support for many-sorted logic is possible to implement syntactically (e.g., by treating sorts as predicates and implicitly introducing axioms that define disjoint universes), however, this would not result in the performance gains we observed.

In addition to unsorted and many-sorted logic, there exists order-sorted logic (Goguen and Meseguer 1992). Order-sorted logic defines a partially ordered set of sorts, and the universes that correspond to these sorts are such that universe of class c_1 is a subset of the universe of class c_2 if $c_1 \leq c_2$. While order-sorted logic is highly similar to our data-models involving multiple inheritance, we are not aware of theorem provers that support it in first order logic with free quantification.

7 Conclusion

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 from a given application and then translating verification queries about the data model to FOL formulae and then using a FOL theorem prover.

We investigated the differences between first order logic (FOL) variants used by theorem provers and the implications of these differences on data model verification. We identified two major differences: (1) the treatment of the type system, and (2) the possibility of empty structures satisfying a given FOL theorem. We formally defined these differences and devised encodings that reconcile them for the purposes of data model verification.

After implementing translations based on these encodings we observed that Z3, an SMT solver, outperformed Spass, a FOL theorem prover, on almost all fronts. Using a many-sorted logic translation that targets Z3, we were able to increase the verification performance by two orders of magnitude, while decreasing the number of inconclusive results by one order of magnitude. With further experiments we showed that encoding our type system using sorts is the cause of this improvement.

References

Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hahnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. **4**(1), 32–54 (2005)

Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: Vericon: towards verifying controller programs in software-defined networks. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), pp. 282–293. ACM, New York (2014)

Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO 2005), pp. 364–387 (2005)

Blanchette, J.C., Popescu, A., Wand, D., Weidenbach, C.: More SPASS with isabelle—superposition with hard sorts and configurable simplification. In: Interactive Theorem Proving—Third International Conference (ITP 2012), Princeton, NJ, USA, August 13–15, 2012. Proceedings, pp. 345–360 (2012)

Bocic, I.: Data model verification via theorem proving. PhD thesis, University of California, Santa Barbara, Sept. 2016

Bocic, I., Bultan, T.: Inductive verification of data model invariants for web applications. In: 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, India—May 31–June 07, 2014, pp. 620–631 (2014)

Bocic, I., Bultan, T.: Coexecutability for efficient verification of data model updates. In: 37th International Conference on Software Engineering (ICSE 2015) (2015a)

Bocic, I., Bultan, T.: Data model bugs. In: NASA Formal Methods—7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27–29, 2015, Proceedings, pp. 393–399 (2015b)

Bocic, I., Bultan, T.: Efficient data model verification with many-sorted logic. In: 30th IEEE/ACM International Conference on Automated Software Engineering ASE 2015, Lincoln, Nebraska, USA, 9–13 Nov. (2015c)

Bocic, I., Bultan, T.: Symbolic model extraction for web application verification. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017, pp. 724–734 (2017)

Claessen, K., Lilliestrom, A., Smallbone, N.: Sort it out with monotonicity—translating between many-sorted and unsorted first-order logic. In: Automated Deduction—CADE-23–23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31–August 5, 2011. Proceedings, pp. 207–221 (2011)

de Moura, L., Bjrner, N.: Efficient e-matching for SMT solvers. In: Automated Deduction—CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17–20, 2007, Proceedings, volume 4603 of Lecture Notes in Computer Science, pp. 183–198. Springer, Berlin (2007)

de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings, pp. 337–340 (2008)

Deutsch, A., Vianu, V.: WAVE: automatic verification of data-driven web services. IEEE Data Eng. Bull. **31**(3), 35–39 (2008)

Deutsch, A., Sui, L., Vianu, V., Zhou, D.: A system for specification and verification of interactive, data-driven web applications. In: SIGMOD Conference, pp. 772–774 (2006)

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

Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17–20, 2006, Proceedings, pp. 81–94 (2006)

Karaca, E.: A collection/list of awesome projects, sites made with Rails, Jan. 2016. https://github.com/ekremkaraca/awesome-rails

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

Fielding, R.T.: Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine (2000)

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

Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: Dynalloy: upgrading alloy with actions. In: 27th International Conference on Software Engineering (ICSE 2005), 15–21 May 2005, St. Louis, Missouri, USA, pp. 442–451 (2005)

Frias, M.F., Pombo, C.L., Galeotti, J.P., Aguirre, N.: Efficient analysis of DynAlloy specifications. ACM Trans. Softw. Eng. Methodol. **17**(1), 4:1–4:34 (2007)

Galeotti, J.P., Frias, M.F.: Dynalloy as a formal method for the analysis of java programs. In: Software Engineering Techniques: Design for Quality, SET 2006, October 17–20, 2006, Warsaw, Poland, pp. 249–260 (2006)

Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. **105**(2), 217–273 (1992)

Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. (TOSEM 2002) **11**(2), 256–290 (2002)
Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: a framework for verifying java programs. In: Computer Aided Verification—28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016, Proceedings, Part I, pp. 352–358 (2016)
Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013), Saint Petersburg, Russia, July 13–19, 2013, pp. 1–35 (2013)
Krasner, G.E., Pope, S.T.: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. J. Object Oriented Program. **1**(3), 26–49 (1988)
Kuncak, V., Lam, P., Zee, K., Rinard, M.C.: Modular pluggable analyses for data structure consistency. IEEE Trans. Softw. Eng. **32**(12), 988–1005 (2006)
Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Proceedings of the 16th International Conference on Logic Programming, Artificial Intelligence, and Reasoning (LPAR), pp. 348–370 (2010)
Lesani, M., Millstein, T.D., Palsberg, J.: Automatic atomicity verification for clients of concurrent data structures. In: Computer Aided Verification—26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings, pp. 550–567 (2014)
Near, J.P., Jackson, D.: Rubicon: bounded verification of web applications. In: Proceedings of the ACM SIGSOFT 20th International Symposium on Foundations of Software Engineering (FSE 2012), pp. 60:1–60:11 (2012)
Nijjar, J., Bultan, T.: Bounded verification of Ruby on Rails data models. In: Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA 2011), pp. 67–77 (2011)
Nijjar, J., Bultan, T.: Unbounded data model verification using SMT solvers. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), pp. 210–219 (2012)
Nijjar, J., Bocić, I., Bultan, T.: An integrated data model verifier with property templates. In: Proceedings of the ICSE Workshop on Formal Methods in Software Engineering (FormaliSE 2013), pp. 23–35. IEEE (2013)
Object Management Group: UML Specification. http://www.omg.org
Open Source Rails, Jan. 2016. http://www.opensourcerails.com
Pelletier, F., Sutcliffe, G., Suttner, C.: The development of CASC. AI Commun. **15**(2–3), 79–90 (2002)
Quine, W.V.: Quantification and the empty domain. J. Symb. Log. **19**(3), 177–179 (1954)
Richters, M., Gogolla, M.: Validating UML models and OCL constraints. In: Proceedings of the 3rd International Conference on Unified Modeling Language (UML 2000), LNCS 1939 (2000)
Ruby on Rails, Feb. 2013. http://rubyonrails.org
SimilarTech: Website technology detection and tracking, Oct. 2018. https://similartech.com/
SMT-LIB, 2016. http://www.smtlib.org/
Spring Framework | SpringSource.org, Feb. 2013. http://www.springsource.org
Stickel, M.E., Waldinger, R.J., Lowry, M.R., Pressburger, T., Underwood, I.: Deductive composition of astronomical software from subroutine libraries. In: Automated Deduction—CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26–July 1, 1994, Proceedings, pp. 341–355 (1994)
Sutcliffe, G., Suttner, C.: The state of CASC. AI Commun. **19**(1), 35–48 (2006)
Sutcliffe, G., Suttner, C.B., Yemenis, T.: The TPTP problem library. In: Automated Deduction—CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26–July 1, 1994, Proceedings, pp. 252–266 (1994)
Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11–15, 2012. Proceedings, pp. 406–419 (2012)
The Web framework for perfectionists with deadlines | Django, Feb. 2013. http://www.djangoproject.com
TPTP Syntax, Jan. 2015. http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html
Tracks, Sept. 2013. http://getontracks.org
Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Boston (1998)

Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Proceedings of the 22nd International Conference on Automated Deduction (CADE 2009), LNCS 5663, pp. 140–145 (2009)

Weidenbach, C.: SPASS input syntax version 1.5, 2016. http://www.spass-prover.org/download/binaries/spass-input-syntax15.pdf

Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08), pp. 349–361. ACM, New York, NY (2008)

Affiliations

Ivan Bocić[1] · Tevfik Bultan[2] · Nicolás Rosner[2]

Ivan Bocić
bocic.ivan@gmail.com

Tevfik Bultan
bultan@cs.ucsb.edu

1 Google, Inc., Mountain View, CA, USA

2 University of California, Santa Barbara, USA