Efficient Data Model Verification with Many-Sorted Logic

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

***Abstract*—Misuse or loss of web application data can have catastrophic consequences in today's Internet oriented world. Hence, verification of web application data models is of paramount importance. We have developed a framework for verification of web application 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. In this paper, 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 experiments demonstrate that using many-sorted logic improves orders of magnitude, and completely eliminates inconclusive results in all cases over 7 real world web applications, down from an 17% inconclusive rate.**

I. Introduction

Modern software applications have migrated from the desktop onto to the cloud. Benefits of web applications over desktop applications include availability 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 complex, distributed systems. These applications typically serve to store and manage user data. For web applications such as Healthcare.gov, Gmail, Twitter and Facebook, loss or corruption of data would have dire consequences. Hence, verification of how applications manage data is of paramount importance.

We focus on a specific but widespread type of web application that is enforced by many popular web application frameworks (Ruby on Rails [30] for Ruby, Spring [33] for Java, Play [28] for Scala and Java, Django [9] for Python, Sails [32] for NodeJS etc.). These web applications employ the Model-View-Controller [21] (MVC) pattern to make web application development easier and modular. 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*. These web applications have the following characteristics: 1) They are RESTful [13], 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. Based on these characteristics we can verify invariants about the data model by analyzing the object oriented code that defines the actions, and verifying that each action preserves the invariants.

Previously, we developed a framework for verification of web application data models by automatically translating verification queries to logic formulas and then using an automated theorem prover to check these automatically generated formulas [3], [4]. In this framework, we first statically extract a semantic data model from the web application by analyzing the data model schema and the methods that implement the actions. 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. More precisely, 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. Then, we send this theorem to an off-the-shelf FOL theorem prover to verify. Using this method on several real world web applications, we discovered several previously unknown bugs that the developers acknowledged and repaired [3].

Since FOL is in general undecidable, an automated theorem prover may never terminate deducing, continuously producing new deductions without reaching a proof of the theorem. This is especially the case in the presence of complex constructs in actions such as loops [4]. Our approach results in one of three outcomes for each action-invariant pair: 1) a conclusive 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 theorem prover we used in our experiments (Spass [41]). 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 deducing 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.

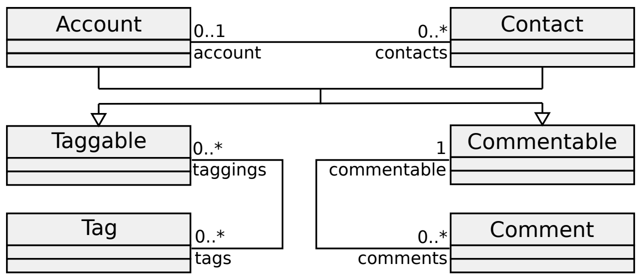
In addition to dealing with inheritance, there is one more complication in translation to many-sorted logic. Classical FOL (i.e., FOL without sorts) defines structures (instances that may or may not satisfy a given set of formulas) as strictly non-empty. In our data model encoding, empty structures represent 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.

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 a total of 3184 verifiable action-invariant pairs 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 in all cases, successfully verifying all these action-invariant pairs whereas Spass had an inconclusive result rate of 17%. 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 reran our experiment suite on Z3 with this unsorted translation. We found that the unsorted translation induced a 3.67% inconclusive result rate, and slowed down verification by two orders of magnitude. While Z3 and Spass use fundamentally different approaches to theorem proving, we posit that sorts are inherently useful in data model verification.

The rest of the paper is organized as follows. Section II presents an overview of our approach in greater detail through examples. Section III defines first order logic, focusing on the differences between unsorted, many-sorted and empty logics.



(a) Data model schema.

```
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
```

(b) Destroy Comment action.

Fig. 1. A data model example based on FatFreeCRM [12].

Section IV defines the type system of data models. Section V explains the translation of data models into different variants of first order logic. In Section VI we evaluate how use of these different logics affect verification feasibility and performance. Section VII discusses related work, and Section VIII concludes the paper.

II. Overview

FatFreeCRM [12] is an open source web application for customer relationship management written in Ruby on Rails. Figure 1 contains an excerpt from the data model of this application. Figure 1(a) defines a portion of the data model schema. Among other things, FatFreeCRM manages Account objects, which can have any number of Contacts. Both Accounts and Contacts are Taggable and Commentable. Even though Taggable and Commentable are implemented using mixins, we treat them as normal classes in our abstraction. Commentable objects have any number of Comments, and every Comment is attached to a Commentable. Taggable objects can have any number of Tags, and Tags can be attached to any number of Taggables.

Data models also specify behaviors via actions, i.e., they specify how the data are modified. Figure 1(b) shows an example action from the FatFreeCRM. This action is taken from the `CommentsController` and it deletes the comment whose `id` is provided by the user. Line 4 reads this identifier from the request (`params[:id]`), looks up the comment for this `id` (using `Comment.find(...)`), and assigns the return value to a variable called `@comment`. Line 5 deletes this comment and line 6 synthesizes the response.

In our framework data models are verified by synthesizing formulas in first order logic (FOL) in order to answer the following question: After defining the model schema, and assuming that all invariants about the data model hold before the action, and taking into account all possible behaviors of the action, do the invariants still hold after the action's execution? The generated FOL formula is valid if and only if the answer to this question is affirmative.

For example, Figure 2 presents a translation of the data model schema from Figure 1(a) to FOL. We define 6 predicates for this purpose: Account, Contact, Taggable, Commentable, Tag, and Comment. We define that all Accounts are Taggable and Commentable (Formula (1)) and that all Contacts are Taggable and Commentable (Formula (2)).

It is useful to define a predicate that denotes that an object is precisely of a class and not of a subclass. This is not trivial to express since, for example, Accounts are simultaneously Taggable but the reverse might not be the case. To that end, we introduce 2 predicates: Taggable_x and $\text{Commentable}_\text{x}$. We define these predicates in Formulas (3) and (4). With these predicates we can define that every object is an instance of at most one class (Formulas (5)-(10)). These formulas imply that, for example, an object could be simultaneously Taggable and Commentable, but not Commentable and a Comment.

Predicates: Account, Contact, Taggable, Commentable, Tag, Comment, Taggable_x, $\text{Commentable}_\text{x}$.

$$\begin{aligned}
&\forall o\colon\ \text{Account}(o) \rightarrow \text{Taggable}(o) \land \text{Commentable}(o) && (1)\\
&\forall o\colon\ \text{Contact}(o) \rightarrow \text{Taggable}(o) \land \text{Commentable}(o) && (2)\\
&\forall o\colon\ \text{Taggable}_\text{x}(o) \leftrightarrow \\
&\qquad \text{Taggable}(o) \land \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) && (3)\\
&\forall o\colon\ \text{Commentable}_\text{x}(o) \leftrightarrow \\
&\qquad \text{Commentable}(o) \land \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) && (4)\\
&\forall o\colon\ \text{Account}(o) \rightarrow \neg\,\text{Contact}(o) \land \neg\,\text{Taggable}_\text{x}(o) \land \\
&\qquad \neg\,\text{Commentable}_\text{x}(o) \land \neg\,\text{Tag}(o) \land \neg\,\text{Comment}(o) && (5)\\
&\forall o\colon\ \text{Contact}(o) \rightarrow \neg\,\text{Account}(o) \land \neg\,\text{Taggable}_\text{x}(o) \land \\
&\qquad \neg\,\text{Commentable}_\text{x}(o) \land \neg\,\text{Tag}(o) \land \neg\,\text{Comment}(o) && (6)\\
&\forall o\colon\ \text{Taggable}_\text{x}(o) \rightarrow \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) \land \\
&\qquad \neg\,\text{Commentable}_\text{x}(o) \land \neg\,\text{Tag}(o) \land \neg\,\text{Comment}(o)) && (7)\\
&\forall o\colon\ \text{Commentable}_\text{x}(o) \rightarrow \neg\,\text{Account}(o) \lor \neg\,\text{Contact}(o) \land \\
&\qquad \neg\,\text{Taggable}_\text{x}(o) \land \neg\,\text{Tag}(o) \land \neg\,\text{Comment}(o) && (8)\\
&\forall o\colon\ \text{Tag}(o) \rightarrow \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) \land \\
&\qquad \neg\,\text{Taggable}_\text{x}(o) \land \neg\,\text{Commentable}_\text{x}(o) \land \neg\,\text{Comment}(o) && (9)\\
&\forall o\colon\ \text{Comment}(o) \rightarrow \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) \land \\
&\qquad \neg\,\text{Taggable}_\text{x}(o) \land \neg\,\text{Commentable}_\text{x}(o) \land \neg\,\text{Tag}(o) && (10)
\end{aligned}$$

Fig. 2. Axioms defining the class diagram in Figure 1(a) in classical (unsorted) first order logic.

The fact that these 10 formulas need to be explicitly stated just to implement these types in FOL demonstrates the semantic gap between the object oriented paradigm and FOL. Generating these formulas from a data model is doable, however, they indicate a potential performance problem during theorem proving. Since the theorem prover does not treat types of variables as integral parts of them, every formula that the theorem prover deduces will need to be additionally processed to deduce the possible types of involved entities. And indeed, upon inspecting the deduction logs of the theorem prover we use (Spass), we found that the theorem prover frequently takes steps to deduce types of entities.

Deduction about types is largely unnecessary in our case. When constructing formulas that define actions and invariants we have specific types in mind. We would be able to provide this type information explicitly if the underlying logic supported type declarations. This would lessen the burden of the theorem prover and possibly drastically reduce the number of deductions necessary.

Many-sorted logic is a variant of FOL that supports explicit type information. In addition, there exist automated theorem provers that accept many-sorted logic and, as our experiments demonstrate, can use the sort information to increase verification performance and viability.

Many-sorted logic supports *sorts*, or types, that are associated with all predicates, functions and variables. Sorts imply mutually disjoint sets of elements (e.g., an element of sort A and an element of sort B can never be equal). Many-sorted logic does not support subsorts, and every domain element is always of a single sort. A formal definition is given in Section III.

We developed a translation of data models to many-sorted logic. Figure 3 demonstrates how the data model in Figure 1(a) would be specified within the many-sorted logic. Since sorts imply mutually exclusive types, and classes Account, Contact, Commentable and Taggable are related through inheritance, we cannot use sorts alone to define this data model. Instead, we introduce a single sort (called Cluster) and four unary predicates Account, Contact, Commentable and Taggable that accept an argument of this sort. We will use these predicates to refer to these classes directly. On the other hand, Tag and Comment are classes that are unrelated to others through inheritance, and as such we can use sorts alone to denote their types.

Formulas (1) and (2) specify that all Cluster objects are Commentable and Taggable. Note that this could have been specified in a simpler way, but the example corresponds to the result of our method presented in Section V. Formulas (3) and (4) define predicates Taggable_x and $\text{Commentable}_\text{x}$. Formulas (5)-(8) define that, within the Cluster, no object can be an instance of more than one type.

Sorts: Cluster, Tag, Comment.
Predicates: Account(Cluster), Contact(Cluster), Taggable(Cluster), Taggable_x(Cluster), Commentable(Cluster), $\text{Commentable}_\text{x}$(Cluster).

$$\begin{aligned}
&\forall\,\text{Cluster}\ o\colon\ \text{Account}(o) \rightarrow \text{Taggable}(o) \land \text{Commentable}(o) && (1)\\
&\forall\,\text{Cluster}\ o\colon\ \text{Contact}(o) \rightarrow \text{Taggable}(o) \land \text{Commentable}(o) && (2)\\
&\forall\,\text{Cluster}\ o\colon\ \text{Taggable}_\text{x}(o) \leftrightarrow \\
&\qquad \text{Taggable}(o) \land \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) && (3)\\
&\forall\,\text{Cluster}\ o\colon\ \text{Commentable}_\text{x}(o) \leftrightarrow \\
&\qquad \text{Commentable}(o) \land \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) && (4)\\
&\forall\,\text{Cluster}\ o\colon\ \text{Account}(o) \rightarrow \\
&\qquad \neg\,\text{Contact}(o) \land \neg\,\text{Taggable}_\text{x}(o) \land \neg\,\text{Commentable}_\text{x}(o) && (5)\\
&\forall\,\text{Cluster}\ o\colon\ \text{Contact}(o) \rightarrow \\
&\qquad \neg\,\text{Account}(o) \land \neg\,\text{Taggable}_\text{x}(o) \land \neg\,\text{Commentable}_\text{x}(o) && (6)\\
&\forall\,\text{Cluster}\ o\colon\ \text{Taggable}_\text{x}(o) \rightarrow \\
&\qquad \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) \land \neg\,\text{Commentable}_\text{x}(o) && (7)\\
&\forall\,\text{Cluster}\ o\colon\ \text{Commentable}_\text{x}(o) \rightarrow \\
&\qquad \neg\,\text{Account}(o) \land \neg\,\text{Contact}(o) \land \neg\,\text{Taggable}_\text{x}(o) && (8)
\end{aligned}$$

Fig. 3. Axioms defining the class diagram in Figure 1(a) in many-sorted logic.

The translation in Figure 3 may not seem like an improvement over the translation in Figure 2. However, not only is the part defining exact type exclusivity shorter in case of many-sorted logic (4 instead of 6 axioms), but these axioms are also shorter (4 instead of 6 conjuncts implied). Also, the difference between the two translations increases if inheritance is rarely used. In case of no inheritance in the data model, no axioms are necessary to model the data model schema. 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. We further discuss these differences in Section V.

There exists another problem with data model verification using FOL. In FOL, a *structure* can loosely be defined as an instance that may satisfy a given FOL formula. Classical FOL is defined in such a way that the structure cannot be empty (more precisely defined in Section III-C). This is not suitable for our purposes. An empty structure corresponds to no objects existing in the data store, which is a possible state that we want to be considered by the theorem prover. We remedy this by introducing new predicates that enable us to define a type system that allows empty domains (discussed in Section V-B).

III. First Order Logic

In this section we give an overview of first order logic (FOL). Definitions in this section are based on [11], but simplified where possible for brevity. After discussing classical FOL, we describe two variants of FOL called many-sorted logic (Section III-B) and empty logic (Section III-C).

A. First Order Logic

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 $f(t_1, t_2 \ldots t_k)$ where $f \in F$ and $t_1 \ldots t_k$ are terms and k is the arity of function f.

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

- $p(t_1, \ldots t_k)$, where $p \in P$ is a predicate of arity k and $t_1 \ldots t_k$ are terms
- $\forall v\colon f$, 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 $f(t_1 \ldots t_k)$ is mapped to the return value of f^U when using elements of U assigned to terms $t_1 \ldots t_k$ as arguments. Similarly, $p(t_1, \ldots t_k)$ is considered to be true if and only if elements corresponding to $t_1 \ldots t_k$ form a tuple that is in P^U. Boolean connectives and equality are interpreted in a standard way. Universal quantification is a bit more involved: $\forall v\colon f$ 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 $\forall x\colon (\forall y\colon x = y)$ 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, $\forall x\colon x = y$ has one free variable y. Since the theorem provers we use do not allow free variables, from this point on, we will only consider formulas without free variables.

B. 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 allows 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 previously, L includes a set of sorts S. Functions and predicates in F and P respectively 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 [5]. 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.

C. Empty Logic

FOL universes are usually defined to be non-empty. Allowing the special case of an empty universe makes definitions more complicated. Treatment of variables and function return values becomes problematic because terms are expected to 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 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, other data model verification tools, such as Alloy [17], support empty domains.

Empty logic is a variant of FOL that allows empty universes. The treatment of empty universe in empty logic is defined by Quine [29]: 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.

IV. Data Models

In this section we formally define our data models. This definition is an extension of the model defined in our previous work [3]. This extended formulation supports multiple inheritance among data model classes.

a) Data Stores: A data store is a structure $DS = \langle C, R, A, I \rangle$ where C is a set of classes, R is a set of relations, A is a set of actions, and I is a set of invariants.

The set of classes C identifies the types of objects that can be stored in the data store. Each class can have any number of superclasses from C ($\text{superclass}(c) \subset C$) and, transitively, no class can be a superclass of itself. We use $c_1 < c_2$ ($c_1 \leq c_2$) to denote that c_1 is a subclass of (or equal to) c_2. A relation $r = \langle n, c_o, c_t, card \rangle \in R$ is defined by a name n, origin class $c_o \in C$, a target class $c_t \in C$ and a cardinality constraint $card$ (such as one-to-one, one-to-many etc.).

b) Data Store States: Given a data store $DS = \langle C, R, A, I \rangle$, the set of all possible *data store states* is denoted as $\overline{DS}$. A data store state is a structure $\langle O, T \rangle \in \overline{DS}$ where O is a set of *objects* and T is a set of *tuples*. Objects are instances of classes, whereas tuples are instances of relations. Each object $o \in O$ is an instance of a class $c \in C$ denoted by $c = \text{classof}(o)$. Each tuple $t \in T$ is in the form $t = \langle r, o_o, o_t \rangle$ where $r = \langle n, c_o, c_t, card \rangle \in R$ 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. Cardinality constraints of each relation $r \in R$ must be satisfied by every data store state in $\overline{DS}$.

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

For example, the action in Figure 1(b) could be defined as follows. We would define that there exists a Comment that is going to be deleted: this Comment will not exist in the post-state, regardless of whether it existed in the pre-state. All associations of this deleted Comment are deleted as well. This alone is insufficient, as this definition allows a behavior where any other object is created or deleted as well. In order to define that this one Comment is the only object affected by the action, we explicitly define that all other Comments exist in the post state if and only if they existed in the pre state. Similarly, for all objects of other types, we define that they exist in the post-state if and only if they existed in the pre-state.

d) Invariants: Given a data store $DS = \langle C, R, A, I \rangle$, I is the set of invariants. An invariant $i \in I$ corresponds to a Boolean function $i \colon \overline{DS} \to \{\text{false}, \text{true}\}$ that identifies the set of data store states which satisfy the invariant. We allow the user to specify invariants in Ruby on Rails directly using a library that we provide that supports quantification. Invariants are translated to FOL almost verbatim.

V. Logics and the Translation

In this section we present the translation of data models (as defined in Section IV) into different variants of first order logic (FOL). The specifics of the underlying logic (many-sorted or not, empty or non-empty) are closely related to the way we encode type system in FOL, as well as the way we quantify over objects in the translation. Discussing how we handle the data model schema and quantification is sufficient to explain and differentiate our translation to different FOL variants. The rest of the FOL translation is described in detail in our earlier work [3] and will not be repeated here due to space constraints.

For brevity, we restrict the following discussion to classes only. Although the definition of relations does differ based on the underlying logic, they are handled in a way analogous to how we handle classes.

Furthermore, 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).

A. Unsorted vs. Many-sorted

In order to define the class system, for both unsorted and many-sorted logics 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. In this section we assume to be given a data store $DS = \langle C, R, A, I \rangle$.

a) Unsorted Logic: Inheritance axioms for unsorted logic are defined as follows. For each class $c \in C$ that has a non-empty superclass set $\text{superclass}(c) = \{p_1, p_2 \dots p_k\}$ 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 Figure 1(a) this method produces Formulas (1) and (2) in Figure 2.

Instance axioms constitute one axiom per class $c \in C$ and serve to define *instance predicates* $\overline{c}_x$. These predicates are used to express that an object is an instance of class c, or more explicitly, 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:

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

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 Figure 1(a) this creates Formulas (3) and (4) in Figure 2.

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:

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

These formulas correspond to Formulas (5)-(10) in Figure 2.

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

b) Many-sorted Logic: Within our translation where universe elements correspond to objects, sorts naturally serve the purpose similar to classes. However, sorts imply disjoint object sets, 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 $c_1, c_2, \dots c_k$ 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 $\text{superclass}(c) = \{p_1, p_2 \dots p_k\}$:

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

For the model presented in Figure 1(a), inheritance axioms are Formulas (1) and (2) in Figure 3.

An instance axiom is generated for each class c. Let $\{s_1 \dots s_k\}$ 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 Figure 1(a), instance axioms are Formulas (3) and (4) in Figure 3.

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 $\{c_1, \dots c_k\}$ 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 Figure 3 correspond to membership axioms for the model in Figure 1(a).

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.

B. 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 above 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.

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 $\exists v\colon \overline{c}(v) \wedge f$.

Predicates: PreState, PostState, AtComment.

$$\forall x\colon \text{AtComment}(x) \Rightarrow \text{Comment}(x) \quad (1)$$
$$\forall x\colon (\forall y\colon \text{AtComment}(x) \wedge \text{AtComment}(y) \Rightarrow x = y) \quad (2)$$
$$\forall x\colon \text{AtComment}(x) \Rightarrow \neg\,\text{PostState}(x) \quad (3)$$
$$\forall x\colon \neg\,\text{AtComment}(x) \Rightarrow (\text{PreState}(x) \Leftrightarrow \text{PostState}(x)) \quad (4)$$

Fig. 4. Unsorted action translation example.

For example, the action presented in Figure 1(b) can be translated to FOL as defined in Figure 4. For brevity, we omit listing all predicates and axioms that define the type system. In this translation, the $\mathrm{AtComment}$ 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 $\mathrm{AtComment}$ is also accepted by $\mathrm{Comment}$ and therefore exists semantically. Next, in Formula (2) we constrain that there exists at most one element in variable $\mathrm{AtComment}$, as the `find` method in Ruby on Rails (line 4 in Figure 1(b)) 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 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 $\mathrm{Comment}$ and $\mathrm{AtComment}$.

This problem becomes more complicated with the many-sorted logic translation. If we were to define a Comment sort alone, 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.

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

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

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

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

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

Fig. 5. Many-sorted action translation example.

Given the example action in Figure 1(b), a many-sorted translation can be defined as in Figure 5. Note that, once again, we omit declaring all sorts, predicates and axioms from Figure 3 for brevity.

Notice that we introduce predicates $\mathrm{Comment}_P$ and Tag_P in addition to previously defined sorts Comment and Tag. In Formula (1) we define that all elements accepted by $\mathrm{AtComment}$ are also accepted by $\mathrm{Comment}_P$. This is necessary to express since, without this axiom, there could be an element accepted by $\mathrm{AtComment}$ that is not accepted by $\mathrm{Comment}_P$. Formula (2) defines that there exists at most one element accepted by $\mathrm{AtComment}$. 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 $\mathrm{Comment}$ 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.

TABLE I. WEB APPLICATIONS USED IN OUR EXPERIMENTS

Application	KLOC	# of Model Classes	# of Action/ Invariant Pairs
Copycopter	2.33	6	21
FatFreeCRM	38.79	34	480
Fulcrum	3.04	5	105
Kandan	6.10	11	20
Lobsters	4.93	13	270
Redmine	87.72	55	2024
Tracks	28.72	39	210

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

VI. Experiments

We conducted two sets of experiments. Both these experiments involved data store verification of 7 real world web applications: Copycopter [6], FatFreeCRM [12], Fulcrum [15], Kandan [19], Lobsters [23], Redmine [31] and Tracks [39]. We extracted the data models from these applications fully automatically [3] and we wrote the invariants to be verified manually. Statistics about these applications are summarized in Table I. Note that KLOC refers to the number of lines of code in Ruby, excluding JavaScript and domain-specific languages.

In addition to these 7 applications, we separately tested our tool on applications and actions that employ loops. In our experience, loops are the most difficult construct to verify, where even the presence of loops with empty iteration bodies may cause timeouts [4]. This data set is useful to test how sorts help with highly complex problems. We included all 54 action/invariant pairs from this data set that used application-specific invariants (and not the trivially preserved *false* invariant).

In total we had 3184 action-invariant pairs. We refer to these pairs as verification cases or verification instances. We translated these 3184 cases into different FOL variants in order to evaluate the performance using different provers, heuristics and translations: a total of 12736 FOL theorems. For each of these cases, we ran verification with a time limit of 5 minutes. If the theorem prover did not deduce a result within 5 minutes 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.

A. FOL Theorem Provers

As an unsorted logic theorem prover we used Spass [41]. 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 [40]. 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* [41] 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

TABLE II. VERIFICATION PERFORMANCE SUMMARY

Method	# of Timeouts		Verification Time (sec)		SMT Unit Propagations (#)		Memory (Mb)		
			Average	Median	Average	Median	Average	Median	Maximum
Spass (Soft sorts on)	558	(17.53%)	5.42	7.89	n/a	n/a	69.00	74.45	129.30
Spass (Soft sorts off)	541	(16.99%)	5.37	7.95	n/a	n/a	72.64	74.45	1686.19
Z3 (Many-sorted)	0	(0.00%)	0.05	0.03	518.77	22.00	3.12	3.47	127.16
Z3 (Unsorted)	117	(3.67%)	1.47	0.24	7704.67	42.00	741.02	33.04	30046.01

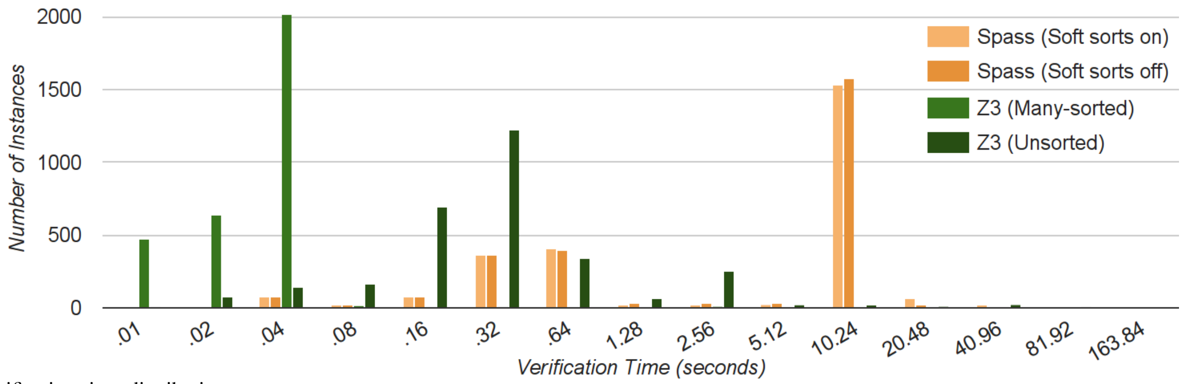


Fig. 6. Verification time distribution.

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 [2]. 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 [8] to evaluate effectiveness of data model verification using many-sorted logic. Z3 is a DPLL(T) [10] based SMT solver that deals with free quantification and uninterpreted functions using E-matching [7].

SMT solvers tend to support many different theories, such as arithmetic, arrays or bitarrays. 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 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.

B. Spass vs. Z3 Performance

Our first set of experiments compare the performance of Spass and Z3 for the purpose of data model verification. These experiments were done solely to detect whether Z3 can sometimes outperform Spass, either by reaching results that Spass could not, or occasionally reaching them faster. 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 II and Figure 6 and 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 SMT Unit Propagations columns. With soft sorts enabled, Spass produced 558 inconclusive results (17.53%). With soft sorts disabled, Spass produced 541 inconclusive results (16.99%). Interestingly, the set of inconclusive results caused by disabling soft sorts is a strict subset of the inconclusive results caused by enabling soft sorts. Spass performed similarly regardless of the soft sorts setting. For both settings, excluding timeouts, verification took an average of over 5 seconds per case. The median case is just under 8 seconds. The memory consumption averaged at around 70Mb, with the median case of 74.45Mb. Disabling soft-sorts caused a peak of 1.6Gb memory used, whereas with soft-sorts enabled, the peak was at 129.30Mb. In only 7 cases did verification with disabled soft-sorts exceed 129.30Mb.

Z3 performed far better than Spass with either heuristic. *Z3 produced no inconclusive results*. On average, Z3 took 0.05 seconds to verify an action-invariant pair. In no case did Spass outperform Z3 in terms of verification time performance. Furthermore, Z3’s average and median memory consumption was under 4Mb, peaking at 127.16. Only for 4 cases did Z3 consume more memory than Spass (with or without soft sorts) and the difference is negligible.

Figure 6 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 seconds 469 times. Spass never achieved a result within this time. The next time range is labeled .02 and shows that Z3 produced a verification result in more than 0.01 seconds but less or equal to 0.02 seconds 641 times.

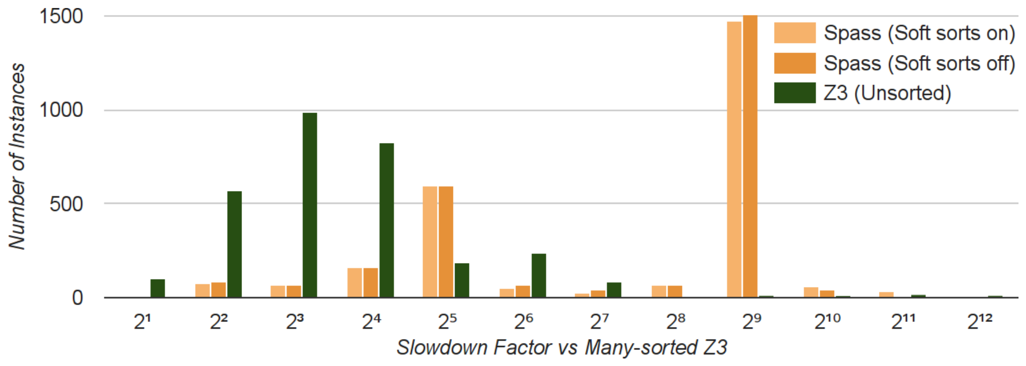


Fig. 7. Distribution of the slowdown factor compared to (many-sorted) Z3.

TABLE III. OBSERVED SLOWDOWN COMPARED TO (MANY-SORTED) Z3

Method	Average	Median	Interdecile Range
Spass (Soft sorts on)	201.40	263.00	11.0 – 274.0
Spass (Soft sorts off)	186.87	265.00	10.5 – 276.7
Unsorted Z3	72.81	8.00	3.7 – 51.3

We wanted to compare the performances of different provers on case-by-case basis. For each action-invariant pair, we calculated the relative slowdown factor induced by a prover compared with Z3. So, for example, if an action-invariant pair was verified 85 times slower using Spass with sorts enabled when compared with Z3, this counts as a slowdown factor of 85. Figure 7 and Table III summarize this data.

Figure 7 contains the distribution of slowdown factors per prover. For example, Spass (with and without soft sorts) is most frequently 2^9 times slower than Z3. Table III contains additional information about this slowdown. On average, Z3 was 201.40 times faster than Spass with soft sorts on, and 186.87 times faster with soft sorts off. In the median case, Z3 was 263 and 265 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 is the range of values ignoring the lowest and highest 10% of values in the sample. The interdecile ranges of performance increases of Z3 over Spass with soft sorts on and off are 11.0-274.0 and 10.5-276.7 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 removed all inconclusive results down from an inconclusive rate of around 17%.

C. Many-sorted vs 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 Section V-Aa. 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 II summarizes the performance of many-sorted and unsorted Z3 verification. Unsorted Z3 did not produce a conclusive result in 117 cases (3.67%). On average, many-sorted Z3 took 0.05 seconds per case whereas unsorted Z3 took 1.47 seconds. Median values are 0.03 for the many-sorted logic and 0.24 for the unsorted translation.

The SMT Propagations columns in Table II refer to the number of DPLL(T) unit propagations done by the prover. On average, the many-sorted translation required 518.77 unit propagations before terminating, with a median number of 22. The unsorted translation required 7704.67 propagations on the average, with a median number of 42. If we were to, for each case, calculate the ratio between unit propagations taken by the unsorted translation over the number of unit propagations taken by the many-sorted translation, we get an average of 41.90 and a median of 1.76. This means that, on average, the unsorted translation took 41.90 more unit propagations to terminate when compared to many-sorted translation.

Finally, the memory footprint of verification suffered as well. The many-sorted translation used an average of 3.12Mb of memory per verification case, with a median of 3.47Mb and peaking at 127.16Mb. The unsorted translation was drastically more demanding, with an average of 741.02Mb, median of 33.04Mb, and peaking at 30046.01 megabytes.

Figure 6 contains data for the unsorted Z3 translation in addition to (many-sorted) Z3 and Spass results. Similarly, Figure 7 and Table III 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 72.81 times faster verification compared to the unsorted translation. The median case is 8.00, and the interdecile range is 3.7-51.3.

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.

Finally, in our experiments, a large majority of actions preserve invariants. This is expected as most actions do not affect data in a way that would invalidate invariants. This is relevant because Z3 is optimized to find satisfying instances (corresponding to falsification in our encoding), and Spass is optimized to prove valid theorems (corresponding to verification in our encoding). There were 59 verification cases that were proven to be invalid (i.e. falsified) by at least one prover. In all 59, Z3 (Many-sorted) proven the case to be invalid successfully, and in all 59, Z3 (unsorted) failed to deduce a result. Spass (Soft sorts on) failed to prove 48 of these, and Spass (soft sorts on) failed to prove 33 of these.

VII. Related Work

This paper's goal is improving the performance and viability of our previous work on data model verification using theorem provers [3], [4]. Our more recent paper [4] looks into verification of loops using a FOL theorem prover (Spass) and how loop verification can be made more viable and better performing. That branch of research is orthogonal to this work. Coexecutability is applicable to both many-sorted and unsorted logic representations regardless of the solution to the empty universe problem.

Verification of software using theorem provers has been explored before in projects such as Boogie [1], Dafny [22] and ESC Java [14]. 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.

Alloy [17], [18] 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.

iDaVer [25], [26], [24] 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.

As part of a research effort to use Spass as the theorem prover engine for interactive theorem proving [2], 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 [20] is a high performance FOL theorem prover that supports sorts. Snark [34] 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 [37], [38]), the syntax of the test suite that is used by the annual World Championship for Automated Theorem Proving [36], [27]. 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 [35], 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 [16]. 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.

VIII. Conclusion

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

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

[2] J. C. Blanchette, A. Popescu, D. Wand, and C. Weidenbach. 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*, pages 345–360, 2012.

[3] I. Bocic and T. Bultan. 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*, pages 620–631, 2014.

[4] I. Bocic and T. Bultan. Coexecutability for efficient verification of data model updates. In *37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[5] K. Claessen, A. Lillieström, and N. Smallbone. 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*, pages 207–221, 2011.

[6] Copycopter, Jan. 2015. http://copycopter.com.

[7] L. de Moura and N. Bjrner. 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*, pages 183–198. Springer, 2007.

[8] L. M. de Moura and N. Bjørner. 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*, pages 337–340, 2008.

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

[10] B. Dutertre and L. M. de Moura. 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*, pages 81–94, 2006.

[11] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

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

[13] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

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

[15] wholemeal: Fulcrum, Jan. 2015. http://wholemeal.co.nz/projects/fulcrum.html.

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

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

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

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

[20] L. Kovács and A. Voronkov. 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.*, pages 1–35, 2013.

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

[22] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 348–370, 2010.

[23] Lobsters, Mar. 2014. https://lobste.rs.

[24] J. Nijjar, I. Bocić, and T. Bultan. An integrated data model verifier with property templates. In *Proceedings of the ICSE Workshop on Formal Methods in Software Engineering (FormaliSE 2013)*, 2013.

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

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

[27] F. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[28] Play Framework - Build Modern & Scalable Web Apps with Java and Scala, May 2015. http://www.playframework.com/.

[29] W. V. Quine. Quantification and the empty domain. *J. Symb. Log.*, 19(3):177–179, 1954.

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

[31] Overview - Redmine, Sept. 2014. www.redmine.org.

[32] Sails.js — Realtime MVC Framework for Node.js, May 2015. http://sailsjs.org/.

[33] Spring Framework — SpringSource.org, Feb. 2013. http://www.springsource.org.

[34] M. E. Stickel, R. J. Waldinger, M. R. Lowry, T. Pressburger, and I. Underwood. 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*, pages 341–355, 1994.

[35] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. 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*, pages 406–419, 2012.

[36] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[37] G. Sutcliffe, C. B. Suttner, and T. Yemenis. The TPTP problem library. In *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, pages 252–266, 1994.

[38] TPTP Syntax, Jan. 2015. http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html.

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

[40] C. Weidenbach. Spass input syntax version 1.5. http://www.spass-prover.org/download/binaries/spass-input-syntax15.pdf.

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