# Data Model Bugs

Ivan Bocić and Tevfik Bultan

Department of Computer Science
University of California, Santa Barbara, USA
bo@cs.ucsb.edu bultan@cs.ucsb.edu

**Abstract.** In today's internet-centric world, web applications have replaced desktop applications. Cloud systems are frequently used to store and manage user data. Given the complexity inherent in web applications, it is imperative to ensure that this data is never corrupted. We overview existing techniques for data model verification in web applications, list bugs discovered by these tools, and discuss the impact, difficulty of detection, and prevention of these bugs.

## 1 Introduction

Software applications have migrated from desktops to the cloud, with many benefits such as, continuous accessibility on a variety of devices, and elimination of software installation, configuration management, updates and patching. These benefits come with a cost in increased complexity.

In order to reduce the complexity and achieve modularity, most modern application frameworks use the Model-View-Controller (MVC) pattern to separate the code for the data model (Model) from the user interface logic (View) and the navigation logic (Controller). Examples of these frameworks include Ruby on Rails (Rails for short), Zend for PHP, Django for Python and Spring for J2EE.

Since modern web applications serve to store and manage user data, the *data model* is a key component of these applications. Data models are responsible for defining the *data model schema,* i.e., the sets of objects and the relations (associations) that describe the stored data format, and *data model actions* which describe the methods used to update the data. For many high profile applications such as HealthCare.gov, DMV, and consumer applications such as Facebook and Gmail, user data is the most valuable asset. Data model correctness is the most significant correctness concern for these applications since erroneous actions can lead to unauthorized access or loss of data.

In this paper we discuss and characterize data model bugs in modern software applications, and the impact of such bugs using concrete examples discovered in open source applications. We show that these bugs have the potential for causing severe and potentially irreparable damage to the data. We discuss the difficulty and plausibility of recovering from these bugs, and survey the known techniques that can help in detecting and preventing these bugs from occurring.

## 2 Data Model Verification Methods

In modern software applications the interactions between the server and the back-end data store is typically managed using an object-relational mapping

This work is supported in part by the NSF grant CCF-1423623.

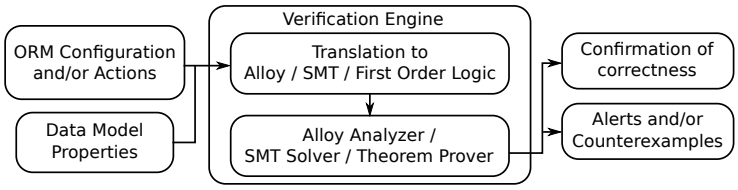


**Fig. 1.** Overview of Data Model Verification

(ORM) library that maps the object-oriented view of the data model at the server side to the relational database view of the data model at the back-end data store. Since the ORM configuration defines the entity types managed by an application, and actions are implemented using the ORM library, investigation of the data model is largely equivalent to investigating the ORM. We overview the results of using three approaches to data model verification, all of which conform to the same overall architecture presented in Figure 1. Note that this is different from verification techniques that are based on formal specifications [2] which impose a semantic gap between the specification and actual source. These techniques do not ensure that the actual implementation conforms to the specification, making them less useful in detecting existing bugs and unable to verify that no bugs exist in the implementation.

For example, investigating the ORM configuration is useful for finding bugs in the data model [5]. This technique translates the ORM configuration (schema) into SMT and Alloy for unbounded and bounded verification. Real bugs were found by using this technique, as well as many misuses of different ORM constructs. However, it has a more limited scope of looking at the configuration only, which allows for false positives in case an invalidation (corruption) is possible in the database, but not possible to create using the actions.

We previously extended this work to include action verification as well [1]. Our technique translates the ORM schema, as well as all the actions, into first order logic. A first order logic theorem prover is then used to verify whether model invariants are preserved by the actions, assuming that actions are executed atomically and in any order (a reasonable assumption for RESTful applications). This technique has found other bugs in real world web applications.

Finally, Rubicon [4] is a library for specifying generalized unit tests. These unit tests are not specified with concrete ORM entities, but with quantification over entity types instead. These tests are automatically translated into Alloy for verification. Using this technique, a security vulnerability was found in a real world web application.

## 3 Reported Data Model Bugs

Before we proceed to discuss data model bugs in general, we will list three web applications and show examples of data model bugs that were found in them [1, 5, 4]. These bugs vary in nature, severity and potential for recovery, presenting useful background for a deeper discussion.

*FatFreeCRM*[1] is an application for customer-relation management. It allows for storing and managing customer data, leads that may potentially become

[1] www.fatfreecrm.com

customers, contacts, campaigns for marketing etc. It spans 30359 lines of code, 30 model classes and 167 actions. Using action verification [1], we found two bugs in FatFreeCRM that we reported to the developers, who confirmed them and immediately fixed one of them. In future discussion we refer to these two bugs as F1 and F2. Rubicon [4] is a tool for verification of Ruby of Rails applications that translates abstract unit tests to Alloy [3], with the goal of ensuring that these tests would pass when given any set of concrete objects. Bug F3, related to access control, was detected using this technique.

Bug F1 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 fundamentally invalid because the application assumes that their owner exists, causing crashes whenever an orphaned `Todo`'s owner is accessed. Because of the severity, this bug was acknowledged and repaired immediately after we submitted a bug report.

Bug F2 relates to `Permission` objects. `Permission` objects serve to define access permissions for either a `User` or a `Group` to a given `Asset`. Our tool has found that it is possible to have a `Permission` without associated `User` or `Group` objects. This bug is replicated by deleting a `Group` that has associated `Permission`s. Although similar to F1 in causality, the repercussions of this bugs are very different. If there exists an `Asset` object whose all `Permission` objects do not 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.

Bug F3 is an access control bug that exposes a `User`'s private `Opportunity` objects to other `User`s. This bug is exploited by registering a new `User` in a way that it shares some of the target `User`'s `Contact`s, giving access to private `Opportunity` objects through these `Contact`s. This bug was caused by a false assumption by the developers that all `Opportunity` and `Contact` objects that belong to the same person will have the same `Permission`s. This bug was reported and acknowledged by the developers.

*Tracks*[2] is an application for organizing tasks, to-do lists etc. This application spans 18023 lines of code, 11 model classes and 70 actions. We identify four bugs in Tracks, which we refer to as T1, T2, T3, and T4. Bug T3 was detected using data model schema verification [5]. Bugs T1, T2 and T4 were discovered using action verification [1], and were reported to and, since, fixed by the developers.

Bug T1 is related to the possibility of orphaning an instance of a `Dependent` class. This bug is similar to bugs F1 and F2, except that the orphaned objects cannot be accessed by actions in any way. Therefore, this bug does not affect the semantics of the application. However, it does present a memory-leak like bug, affecting performance by unnecessarily populating database tables and indexes.

Bug T2 is very similar in nature to T1. When a `User` is deleted, all `Project`s of the `User` are deleted as well, but `Note`s of deleted `Project`s remain orphaned. These orphaned `Note` objects are not accessible in any way, however, the orphaned `Todo`s take up space in the database and inflate indexes.

[2] getontracks.org

Bug T3 is caused by deleting a `Context` without correctly cleaning up related `RecurringTodo` objects. This is similar to bug F1 because the orphaned `RecurringTodo` objects are accessible by the application and cause the application to crash.

We found bug T4 when the action verification method [1] reported an inconclusive result within the action used to create `Dependent` instances between two given `Todo`s. Semantically, there must not be dependency cycles between `Todo`s; this is a structural property of the application. Our method could not prove or disprove that cycles between `Todo`s cannot be created. Upon manual inspection we found that, while the UI prevents this, HTTP requests can be made to create a cycle between `Todo`s. 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 when repairing the state of the data may be impossible, as discussed in Section 4.

*LovdByLess*[3] is a social networking application. It allows people to create accounts, write posts and comment on posts of other users, upload and share images etc. It contains 29667 lines of code, 12 model classes and 100 actions.

Data model schema analysis [5] was used to find bug L1 where the `Comment`s of a `User` were not cleaned up properly when a `User` is deleted. The orphaned `Comment`s, however, remain connected to the `Post` they belong to, and are visible from the said `Post`. The application previews these `Comment`s, along with their content and other data, except for the author. The author's name field remains blank. This is not expected behavior: either the `Comment`s are supposed to be deleted, or they are supposed to remain, in which case the author's data is lost.

## 4 Discussion on Data Model Bugs

We identified two types of bugs: access control bugs and data integrity bugs. Access control bugs give access to data to users with insufficient privileges. Data integrity bugs are bugs that allow invalidation of the application's data. Note that we draw a distinction between bugs that allow data to be invalidated and bugs that are caused by data that has been invalidated. The latter bug is a symptom of the former.

*Severity.* Access control policies are hard to correctly specify and hard to correctly enforce [4]. Access control bugs are severe bugs. Exposing private information is not permissible in any application that stores and manages private information, nor is allowing access to admin or root level operations.

The severity of data integrity bugs varies on the specifics of the bug, spanning from benign bugs that at most cause minor performance problems, over bugs causing crashes in the application, to bugs causing data loss and corruption from which recovery is exceptionally difficult or impossible.

[3] github.com/stevenbristol/lovd-by-less

We identified several data integrity bugs that allow invalid data to exist in the database, but in such a way that this invalid data is never used by the application. We refer to these bugs as *data model leaks*. They are usually caused by incorrect cleanup of related entities when an entity is deleted. This category is demonstrated by bugs T1 and T2. These bugs are hard to detect unless the leaked data accumulates to a certain point. Their impact is limited to performance, not affecting the semantics of the program. They negatively impact performance by taking up space in the database and populating indexes unnecessarily.

In most cases, the corrupted data can be accessed by the application, causing the application to misbehave in some way. We identified a wide range of misbehavior severity. For example, orphaned objects may be visible to the user as empty fields on the webpage (L1), allow operations and further data updates that should not be allowed (F2), or crash the web application (F1, T3, T4).

*Recovery.* Access control bugs allow no recovery. Once private information has been exposed, fixing the bug only prevents future threats. No measure exists to make the exposed information private again. However, data integrity bugs have recovery potential. Repairing a data integrity bug involves two steps: repairing the data and preventing future invalidation.

In some cases, data is recoverable. For example, once a data model leak is discovered, leaked entities can be identified and removed. The same applies in the case of data being incorrectly deleted: bugs F1, F2, T3 are recoverable from because the original intent of the developer was to delete data. Removing the invalid data not only removes the corruption, but also brings the data store to the state that was originally expected by the developers.

Data integrity bugs that do not manifest themselves through improper deletion are far more difficult to recover from. Repairing the corruption implies modifying the corrupted data into valid data, which may be impossible. T4 is an example of a bug in which valid data is not distinguishable from invalid data. Even clearly distinguishable corrupted data may be unrecoverable if, for example, invalid data has overwritten correct data.

Backups can be used to recover corrupted data in certain cases. This would be a manual and error prone effort, however, and it would rollback the user's data to a previous point which may be undesirable. To make matters worse, since data integrity bugs are observable only if the data has already been invalidated, the corruption may have been backed up in the time frame between the cause of the corruption and the escalation of the bug, making backups unusable.

*Detection and Prevention.* Data model bugs are hard to anticipate, and addressing them after being detected by users is undesirable because recovery may be extremely difficult. Detection of access control bugs is difficult since a malicious user may leave no trace when accessing restricted information. Similarly, data integrity bugs are hard to detect. They are not observed until the application accesses the invalidated (corrupted) data and misbehaves, which may not be possible (as is the case with bugs T1 and T2). If a user does access the invalidated data, the resulting faulty behavior cannot be replicated by the developer without being given access to the same invalidated data. Furthermore, even given access

to this data, the code causing this strange behavior may be correct. No trace exists on how the data was originally invalidated.

Runtime validation is a commonly used technique for the prevention of potential data model integrity bugs. We define runtime validation as any runtime check that aborts the operation with the goal of preventing invalidation. In web application frameworks, validation can be done in the web application layer automatically (both Rails and Django support user definable model validators), or could be manually implemented in actions (in form of conditional branches that abort unless a specific condition is met), or in the database by defining constraints. Frequently multiple approaches are used: for example, the database may validate the integrity of foreign keys, whereas the application layer may validate that email strings adhere to a given format. Runtime validation alone provides an insufficient solution to the problem. This is demonstrated by the fact that we found serious bugs in applications that heavily rely on runtime validation, and have attempted enforcing security policies. For all the bugs we found, the problem was caused by incorrect implementation.

Considering the difficulty of detection, potential severity and unrecoverability of these bugs, we strongly believe that automated verification techniques should‘ be used to prevent data model bugs. Besides ensuring that static and runtime constraints are sufficient, verification could also be used to detect unnecessary validation. Supported by verification, runtime validation can be more effective in successfully preventing data model bugs.

## 5 Conclusions

Cloud-based modern software applications store and manipulate their data on remote servers as defined by the data model. We argue that the correctness of the data model is an essential difficulty in building modern software applications. We have demonstrated data model bugs in several open source applications. We have discussed the difficulties in detection, severity, and the potential for recovery from these bugs. Although there have been automated verification techniques proposed for detecting data model bugs, they all have their limitations. Detection and prevention of data model bugs remains to be an important research direction.

## References


1. I. Bocić and T. Bultan. Inductive verification of data model invariants for web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, May 2014.
2. A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *Journal of Computer and System Sciences*, 73(3):442–474, 2007.
3. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Enginnering and Methodology (TOSEM 2002)*, 11(2):256–290, 2002.
4. J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *Proceedings of the ACM SIGSOFT 20th Int. Symp. Foundations of Software Engineering (FSE 2012)*, pages 60:1–60:11, 2012.
5. J. Nijjar. *Analysis and Verification of Web Application Data Models.* PhD thesis, University of California, Santa Barbara, Jan. 2014.