Coexecutability for Efficient Verification of Data Model Updates

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

***Abstract*—Modern applications use back-end data stores for persistent data. Automated verification of the code that updates the data store would prevent bugs that can cause loss or corruption of data. In this paper, we focus on the most challenging part of this problem: automated verification of code that updates the data store and contains loops. Due to dependencies between loop iterations, verification of code that contains loops is a hard problem, and typically requires manual assistance in the form of loop invariants. We present a fully automated technique that improves verifiability of loops. We first define *coexecution*, a method for modeling loop iterations that simplifies automated reasoning about loops. Then, we present a fully automated static program analysis that detects whether the behavior of a given loop can be modeled using coexecution. We provide a customized verification technique for coexecutable loops that results in more effective verification. In our experiments we observed that, in 45% of cases, modeling loops using coexecution reduces verification time between 1 and 4 orders of magnitude. In addition, the rate of inconclusive verification results in the presence of loops is reduced from 65% down to 24%, all without requiring loop invariants or any manual intervention.**

I. Introduction

Nowadays, it is common for software applications to store their persistent data in a back-end data store in the cloud. For many application domains, such as social networking, data is the most valuable asset of an application. Hence, the correctness of the code that updates the data is of significant concern. In this paper, we present automated verification techniques that improve the verifiability of code that update the data.

Typically, application programmers write object oriented code for accessing and updating the data in the back-end data store. This code is automatically translated to data-store queries using Object Relational Mapping (ORM). Modern software development frameworks use the Model-View-Controller (MVC) [23] architecture to separate the user interface code (View) from the code that handles the user requests (Controller), and the code that accesses and modifies the data store (Model). The inherent modularity in this architecture creates opportunities for automated verification. In a typical RESTful [11] MVC-based application, user requests trigger execution of *actions* that read from or write to the back-end data store using an ORM library. Data store related bugs can be eliminated by verifying the implementations of these actions.

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

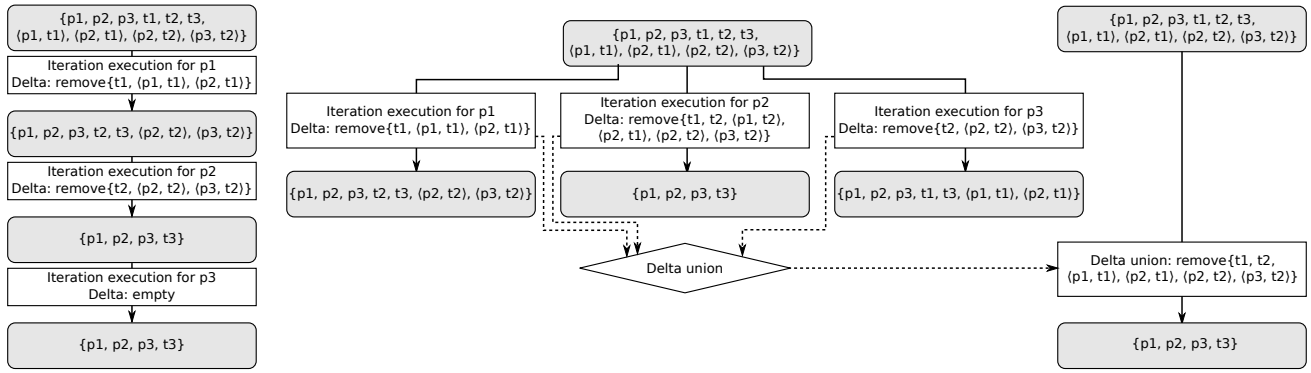
```
1  class PostsController
2    def destroy_tags
3      ...
4      posts = Post.where(id: params[:post_ids])
5      ...
6      posts.each do |p|
7        p.tags.destroy_all!
8      end
9      ...
10   end
11 end
```

Fig. 1. An Example Action

In our earlier work [5], we demonstrated that one can check invariants about the data store by translating verification queries about actions to satisfiability queries in First Order Logic (FOL), and then using an automated FOL theorem prover to answer the satisfiability queries. However, due to undecidability of FOL, an automated theorem prover is not guaranteed to come up with a solution every time, and sometimes it may time-out without providing a conclusive result. In particular, actions that have loops in them are hardest to check automatically. Verification of code that contains loops typically requires manual intervention where the developer has to provide a loop invariant in order to help the theorem prover in reasoning about the loop. This reduces the level of automation in the verification process and, hence, its practical applicability.

In this paper, we present a fully automated technique that significantly improves the verifiability of actions with loops. Our key contribution is the definition of a concept we call *coexecution*, which, while intuitively similar to parallel or concurrent execution, does not correspond to an execution on actual hardware. It is a concept we introduce specifically to make verification easier. We call a loop *coexecutable* if coexecution of its iterations is equivalent to their sequential execution. We present an automated static analysis technique that determines if a loop is coexecutable. We also developed a customized translation of coexecutable loops to FOL that exploits the coexecution semantics and improves verifiability.

We implemented these techniques for verification of actions that update the data model in applications written using the Ruby-on-Rails (Rails) framework [26]. We applied our approach to verification of actions extracted from real world Rails applications. Our experimental results demonstrate that exploiting the coexecutability property significantly improves the verifiability of actions with loops.



(a) Sequential Execution (b) Coexecution

Fig. 2. Sequential execution compared to Coexecution

The rest of the paper is organized as follows. Section II provides an overview of the problem and presents coexecution at an intuitive level. Section III formally defines data stores and sets up the foundation for Section IV, in which we define coexecution and define and prove the coexecutability condition. Section V presents the static program analysis algorithm that decides whether a given loop is coexecutable. Section VI demonstrates the benefits of modeling loops using coexecution on loops extracted from real world applications. Section VII discusses related work, and Section VIII concludes the paper.

II. Overview

Recently, we presented an automated verification framework [5] for verifying the actions that update the data, which 1) translates the action code and user-specified invariants into a formal model we call Abstract Data Store (ADS), 2) translates verification queries about ADS specifications into first order logic (FOL), and 3) verifies them using an off-the-shelf FOL theorem prover [32].

The key obstacle in automated verification of actions is verification of loops. The inclusion of even simple loops in actions greatly increases verification time, and it also increases the chance that the theorem prover times out without ever reaching a conclusion on whether the given action breaks an invariant. Upon manual inspection of the theorem prover deduction logs, we found out that the key issue that the theorem prover struggles with is reasoning about the iteration interdependencies in loops.

In order to verify loops, it is necessary for the automated theorem prover to deduce the rules that define which objects and associations will exist after the loop has executed. To accomplish this, the theorem prover has to compute the transitive closure of the loop body which is not expressible in FOL, hence FOL theorem provers cannot make such deductions. Lacking that capability, the theorem prover attempts to deduce the state of the data store after the first iteration has executed, then after the second iteration has executed, etc. Since we do not bound the number of iterations, unless the theorem prover discovers a way to violate the invariant without reasoning about the loop, which is a very specific case, the automated deduction process employed by the theorem prover does not terminate. This causes the theorem prover to time-out without a conclusive result.

A. Coexecution Overview

Figure 1 presents an example Rails action based on an open source discussion platform called Discourse [7]. This action deletes all the `Tag` objects associated with a set of `Post` objects. The set of `Post` objects that are chosen by the user are assigned to a variable called `posts` in line 4. Then, using a loop (lines 6 to 8) that iterates over each `Post` object in `posts` (with the loop variable `p`), all the `Tag` objects that are associated with the objects in `posts` are deleted.

A data store invariant about the Discourse application could be that each `Tag` object in the data store is associated with at least one `Post` object. In order to verify such an invariant, we need to prove that each action that updates the data store (such as the one shown in Figure 1) preserves the invariant.

As we discussed above, actions with loops are especially hard to verify when translated to FOL. The observation that lead to the technique we present in this paper is, the loops can often be modeled in a way that does not require iteration interdependency. And, in such cases, it is possible to translate the loops to FOL in a way that is more amenable to verification. We call this alternate execution model for loops that removes iteration interdependency *coexecution*.

Consider the action shown in Figure 1. Assume that the initial data store state contains three `Post` objects $p1$, $p2$ and $p3$ and three `Tag` objects $t1$, $t2$ and $t3$, with the following associations: $\langle p1, t1\rangle, \langle p2, t1\rangle, \langle p2, t2\rangle, \langle p3, t2\rangle$. Also, assume that the loop iterates on all the `Post` objects in the order $p1, p2, p3$ (i.e., the variable `posts` is the ordered collection of $p1, p2, p3$).

Given this initial state, the standard sequential execution semantics of the loop in lines 6 to 8 in Figure 1 is shown in Figure 2(a) where gray rectangles with rounded corners denote states of the data store, and solid line arrows denote iteration executions.

The first iteration, executed for `Post p1`, deletes all `Tags` of $p1$ and their associations. We identify the operations executed by this iteration and summarize them as a *delta* of this

operation (i.e., the changes in the data store state caused by this operation). The first iteration removes $t1$ and all associations of $t1$ ($\langle p1, t1\rangle$ and $\langle p2, t1\rangle$). Similarly, the second iteration deletes all `Tags` of $p2$, resulting in a delta that removes $t2$, $\langle p2, t2\rangle$ and $\langle p3, t2\rangle$. Finally, the third iteration does not alter the data store state as $p3$ is not associated with any `Tags` at that point.

Figure 2(b) demonstrates the alternate *coexecution* semantics for the same loop. Instead of executing iterations sequentially to reach the post-state, we first identify the delta for each iteration directly from the pre-state, in isolation from other iterations. As expected, the first iteration (the one for $p1$) is identical to the one from Figure 2(a). However, the iteration for $p2$ deletes all `Tags` of $p2$, its delta removing $t1$ and $t2$ and all their associations. Note that this delta is different from the delta of the sequential execution iteration for $p2$ shown in Figure 2(a). Similarly, the iteration for $p3$ deletes $t2$ and all associations whereas sequential execution for $p3$ produced an empty delta. The deltas of these independent executions are combined together using the *delta union* operation, which in this case returns a union of all the delete operations (we formally define the deltas and the delta union operation in Section IV). In this example, the unified delta removes $t1$, $t2$ and their associations. Finally, we use the unified delta to migrate from the pre-state to the post-state in one step, reaching the same post-state we acquired using sequential execution as shown in Figure 2(b). We call this one step execution semantics based on the unified delta, coexecution.

For some loops, based on the dependencies among loop iterations, coexecution will yield a different result than sequential execution. However, coexecution is equivalent to sequential execution for some classes of interdependencies. Note that, in our example, iterations *are* interdependent (since the $p1$ iteration prevents the $p2$ iteration from deleting $t1$ and its associations), and yet coexecution and sequential execution produce identical results. In Section IV-C, we formally define the *Coexecutability Condition* that, if true for a given loop, guarantees that coexecution of the loop is equivalent to sequential execution of iterations. In Section V we implement this condition as a static program analysis, and based on this analysis we are able to translate loops to FOL in a manner that is more amenable to verification.

In our analysis, we assume that actions that update the data store are executed as transactions (which is the case for most MVC-based web applications). Hence, the database ensures that actions do not interfere with one another during runtime. Effectively, all actions can be considered to execute within atomic blocks, and hence, so are the loops we are verifying. This gives us to freedom to model operations within a loop in any order, as long as the final effects of the loop execution are identical to the effects of sequential execution.

III. Formal Model

In this section we present the formal model we use for modeling data stores.

A. Data Store

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

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

1) Data Store States: Given a data store $DS = \langle C, R, A, I\rangle$, the set of all possible *data store states* is denoted as $\overline{DS}$. Each data store state is a structure $\langle O, T\rangle \in \overline{DS}$ where O is a set of *objects* and T is a set of *tuples*. Together, we refer to objects and tuples as *entities*.

We simplify our notation by using certain operators on objects and tuples indiscriminately. For a state $s = \langle O, T\rangle \in \overline{DS}$, object o and tuple t that may or may not be in s:

$$s \cup \{o\} = \langle O \cup \{o\}, T\rangle \qquad s \cup \{t\} = \langle O, T \cup \{t\}\rangle$$
$$s \setminus \{o\} = \langle O \setminus \{o\}, T\rangle \qquad s \setminus \{t\} = \langle O, T \setminus \{t\}\rangle$$
$$o \in s \Leftrightarrow o \in O \qquad t \in s \Leftrightarrow t \in T$$

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 $r = \langle name, c_o, c_t, card\rangle \in R$ and $\text{classof}(o_o) = c_o$ and $\text{classof}(o_t) = c_t$. For a tuple $t = \langle r, o_o, o_t\rangle$ we refer to o_o as the origin object and o_t as the target object, and $\forall t = \langle r, o_o, o_t\rangle, \forall s \in \overline{DS} : t \in s \Rightarrow o_o \in s \wedge o_t \in s$. Cardinality constraints of each relation $r \in R$ must be satisfied by every data store state in $\overline{DS}$.

Note that, in the abstract data store model, objects do not have basic fields. This model focuses on objects and how they are associated with one another. Basic fields can be introduced to the model by treating them as associations to a basic type class. For example, an `age` field would be an association to one object of the `Integer` class (this way of treating basic fields has been used in Alloy [19], [20] for example). Observe that, in the abstract data store model, all updates consist of creations and deletions of entities (i.e., objects and tuples).

2) Actions and Invariants: Given a data store $DS = \langle C, R, A, I\rangle$, A denotes the set of actions. Each action $a \in A$ corresponds to a set of possible state transitions $\langle\langle O, T\rangle, \langle O', T'\rangle\rangle \subseteq \overline{DS} \times \overline{DS}$. Actions characterize possible updates to data store states (i.e., the transitions between the states).

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

3) Behaviors: Given a data store $DS = \langle C, R, A, I\rangle$, a behavior of a data store DS is an infinite sequence of data store states $s_0, s_1, s_2 \ldots$ where

- $\forall k \geq 0 : s_k \in \overline{DS} \wedge \exists a \in A : \langle s_k, s_k + 1\rangle \in a$
- $\forall k \geq 0, \forall i \in I : i(s_k) = \text{true}$

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

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

B. Statements

An action is composed of a finite number of *statements*. A statement 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, a statement S_c that creates an object of class C could be defined as:

$$[s, s']_{Sc} \leftrightarrow \exists o_c : \text{classof}(o_c) = C \wedge \\ \left(o_c \in s' \wedge o_c \notin s \wedge \left(\forall e : (e \in s \leftrightarrow e \in s') \vee e = o_c\right)\right)$$

As an other example, consider a `Block` statement B which is a sequence of statements A_i, for $1 \leq i \leq n$ for some n. Statement A_1 transitions 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:

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

1) ForEach Loop Statement: A `ForEach` loop statement (FE) is defined by two parameters: the set of objects being iterated over, denoted as α, and the block of code B that will be executed for each member of α. Let $|\alpha| = n$. The order of iteration is non-deterministic. B has access to the iterated object and, therefore, the set of possible executions of B is 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 other words, a pair of states is an execution of a given loop FE if and only if there exists a selection of objects from α and a sequence of states such that (1) the said selection 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 (2).

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.

IV. Coexecutability

Automated reasoning about loops is difficult since it is necessary to take into account many possible intermediate states that can appear during the loop execution. A loop invariant can provide a compact characterization of the intermediate loop states and help with automated reasoning, but loop invariant discovery is itself a difficult problem, and loop invariants are typically specified manually. The coexecution concept we introduce in this paper is a novel approach that enables us to reason about the loop behavior without loop invariants and reasoning about intermediate states.

Below, we formally define coexecution. We also give a condition under which coexecution is equivalent to sequential execution, and we call this property coexecutability. Note that multiple iterations of a loop correspond to repeated sequential execution of the loop body. In order to simplify our presentation, we will discuss how any two statements A and B can be coexecuted (which, for example, can represent the execution of the same loop body twice for different values of the iterator variable). This discussion can be extended to coexecution of any number of statements, and, hence, is directly applicable to loops by treating iterations of a loop as separate statements.

In this section, for brevity and simplicity, we will assume that the data store we reason about contains only one class called `Class` and only one relation called `relation` that associates objects of type `Class` with objects of type `Class` with many-to-many cardinality. This allows us to use minimal notation for data-store states, avoiding the need to explicitly provide type information. For example, the state $\{a, b, c, \langle a, b \rangle, \langle a, c \rangle\}$ contains exactly three objects of type `Class`, as well as two tuples of the `relation` type that associate object a with the other two.

As we discussed earlier, given two statements A and B, their sequential composition $A; B$ is defined by the sequential execution formula below:

$$\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_{A;B} \Leftrightarrow \exists s_i : [s, s_i]_A \wedge [s_i, s']_B \quad (3)$$

A. Execution Deltas

In order to define coexecution, we first need to define a way to express the effects of executing statements. Let us define a structure $\langle O_c, T_c, O_d, T_d \rangle$ for that purpose: O_c and T_c are sets of objects and tuples, respectively, that are created by a given execution, and O_d and T_d are sets of objects and tuples, respectively, that are deleted by a given execution. Let us call this structure the *delta* of an execution.

Given an execution from state s to s', we denote the delta of this execution as $s' \ominus s$. For example, if given two states $s_1 = \{a, b, \langle a, b \rangle\}$ and $s_2 = \{a, b, c\}$, then $s_2 \ominus s_1 = \langle \{c\}, \{\}, \{\}, \{\langle a, b \rangle\} \rangle$ and $s_1 \ominus s_2 = \langle \{\}, \{c\}, \{\langle a, b \rangle\}, \{\} \rangle$.

A delta is *consistent* if and only if its corresponding create and delete sets are mutually exclusive, i.e.,

$$O_c \cap O_d = T_c \cap T_d = \emptyset \wedge \\ (\forall t = \langle o_o, o_t \rangle \in T_c : o_o \notin O_d \wedge o_t \notin O_d) \wedge \\ (\forall t = \langle o_o, o_t \rangle \in T_d : o_o \notin O_c \wedge o_t \notin O_c)$$

In order to combine the changes done by different executions, we introduce the union ($\cup$) of two deltas:

$$\forall \delta_1 = \langle O_{c1}, T_{c1}, O_{d1}, T_{d1} \rangle, \delta_2 = \langle O_{c2}, T_{c2}, O_{d2}, T_{d2} \rangle : \\ \delta_1 \cup \delta_2 = \langle O_{c1} \cup O_{c2}, T_{c1} \cup T_{c2}, O_{d1} \cup O_{d2}, T_{d1} \cup T_{d2} \rangle$$

We will use this operation to merge the changes done by independently executed statements. Note that the result of the union operation may not be a consistent delta even if all the arguments were individually consistent. We call deltas *conflicting* if and only if their union is not consistent.

1) Delta Apply Operation: We will introduce the *apply* operation, that takes a state s and a consistent delta δ and updates the state as dictated by the delta. The result is a new state that contains all objects and tuples that existed in s and were not deleted by δ, and all objects and tuples created by δ. In addition, whenever an object is deleted, all the tuples referring to that object are deleted as well. The apply operation maps a state and a consistent delta into a state, and we use the $\oplus$ operator to denote this operation. Formally, given a state s and a consistent delta $\delta = \langle O_c, T_c, O_d, T_d \rangle$:

$$\forall s = \langle O, T \rangle \in \overline{DS}, s' = \langle O', T' \rangle \in \overline{DS} : s' = s \oplus \delta \Leftrightarrow \\ (\forall o : o \in O' \Leftrightarrow (o \in O \vee o \in O_c) \wedge o \notin O_d) \wedge \\ (\forall t = \langle o_o, o_t \rangle : t \in T' \Leftrightarrow (t \in T \vee t \in T_c) \wedge \\ t \notin T_d \wedge o_o \in O' \wedge o_t \in O')$$

For example, given a state $s = \{a, b, c, \langle a, b \rangle\}$ and a delta $\delta = \langle \{c\}, \{b\}, \{\langle a, c \rangle\}, \{\} \rangle$, $s \oplus \delta = \{a, c, \langle a, c \rangle\}$. Notice how the creation of object c was idempotent given that s already had that object, and that deletion of object b implied that all tuples related to b were deleted as well.

We can observe that $\forall s, s' \in \overline{DS} \times \overline{DS} : s' = s \oplus (s' \ominus s)$. This follows directly from the definition, as $s' \ominus s$ will create all entities (objects and tuples) in s' that are not in s and delete all the entities that are part of s and not s'.

Lemma 1: Given any two non-conflicting deltas δ_1 and δ_2:

$$\forall s \in \overline{DS} : (s \oplus \delta_1) \oplus \delta_2 = s \oplus (\delta_1 \cup \delta_2)$$

This lemma follows directly from definitions of delta union and the apply operation. For simplicity we will limit the proof to objects, but the same proof can be extended to cover tuples.

Given a state $s = \langle O, T \rangle$, non-conflicting deltas $\delta_1 = \langle O_{c1}, T_{c1}, O_{d1}, T_{d1} \rangle$ and $\delta_2 = \langle O_{c2}, T_{c2}, O_{d2}, T_{d2} \rangle$, and post-states $s_s = \langle O_s, T_s \rangle = (s \oplus \delta_1) \oplus \delta_2$ and $s_p = \langle O_p, T_p \rangle = s \oplus (\delta_1 \cup \delta_2)$, we proceed to show that any object in s_s must be in s_p, and that any object in s_p must be in s_s.

$$\forall o \in O_s : (o \in O \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \vee \\ (o \in O_{c1} \wedge o \notin O_{d2}) \vee o \in Oc_2 \\ \Rightarrow \forall o \in O_s : (o \in O \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \vee \\ (o \in O_{c1} \vee o \in Oc_2)$$

Because these deltas are non-conflicting, $(o \in O_{c1} \vee o \in O_{c2}) \Rightarrow (o \notin O_{d1} \wedge o \notin O_{d2})$. Joining this implication with the previous formula:

$$\forall o \in O_s : (o \in O \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \vee \\ ((o \in O_{c1} \vee o \in O_{c2}) \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \\ \Rightarrow \forall o \in O_s : (o \in O \vee o \in O_{c1} \vee o \in O_{c2}) \wedge o \notin O_{d1} \wedge o \notin O_{d2} \\ \Rightarrow \forall o \in O_s : (o \in O \vee o \in O_{c1} \cup O_{c2}) \wedge o \notin O_{d1} \cup O_{d2} \\ \Rightarrow \forall o \in O_s : o \in O_p$$

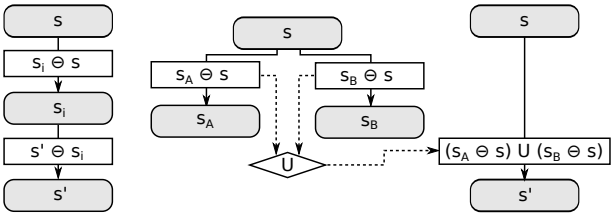


Fig. 3. Sequential execution vs. Coexecution

The inverse implication also holds:

$$\forall o \in O_p : (o \in O \vee o \in O_{c1} \cup O_{c2}) \wedge o \notin O_{d1} \cup O_{d2} \\ \Rightarrow \forall o \in O_p : (o \in O \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \vee \\ (o \in O_{c1} \wedge o \notin O_{d2}) \vee o \in O_{c2} \\ \Rightarrow \forall o \in O_p : o \in O_s$$

A consequence of this property is that the delta apply operation is commutative for non-conflicting deltas (as delta union is trivially commutative).

B. Coexecution

Coexecution of two statements A and B, which we denote as $A|B$, means finding the deltas of independent executions of both statements starting from the pre-state, finding the union of those deltas, and applying the union to the pre-state. This is visualized in Figure 3, similar to Figure 2 but applied to two generic states and any two statements A and B. Formally,

$$\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_{A|B} \Leftrightarrow \exists s_A, s_B \in \overline{DS} \times \overline{DS} : \\ [s, s_A]_A \wedge [s, s_B]_B \wedge s' = s \oplus ((s_A \ominus s) \cup (s_B \ominus s)) \quad (4)$$

Notice that coexecution, because of the delta apply operation, is defined only if no two possible deltas from the pre-state via statements A and B are conflicting.

For example, if statement A adds a new object to a state, and statement B deletes all tuples from a state, executing these statements from the state $s = \{a, b, \langle a, b \rangle\}$ independently will yield the following states:

$$s_A = \{a, b, c, \langle a, b \rangle\}, \quad s_B = \{a, b\}$$

Therefore,

$$s_A \ominus s = \langle \{c\}, \{\}, \{\}, \{\} \rangle \\ s_B \ominus s = \langle \{\}, \{\}, \{\}, \{\langle a, b \rangle\} \rangle \\ (s_A \ominus s) \cup (s_B \ominus s) = \langle \{c\}, \{\}, \{\}, \{\langle a, b \rangle\} \rangle$$

and, the coexecution $A|B$ will result in the following state:

$$s \oplus ((s_A \ominus s) \cup (s_B \ominus s)) = \{a, b, c\}$$

which is the same state to which sequential execution of A and B would transition from s, which means that A and B are coexecutable.

C. Coexecutability Condition

Not all statements are coexecutable since coexecution requires non-conflicting deltas, and even if their deltas are not conflicting, the result of coexecution may not be equal to the result of sequential execution. Below we define a *coexecutability condition*, such that, given any two statements A and B, if A and B satisfy the coexecutability condition, then their sequential execution is always equivalent to their coexecution.

1) Statement Reads, Creates and Deletes: We model each statement as a set of (potentially non-deterministic) state transitions. This definition of statements is very general and widely applicable, but makes it difficult to identify a statement's read set. We need to have access to a statement's read set in order to reason about interdependencies of statements. In the remainder of this subsection we define how to infer a statement's read, create and delete sets from its transition set.

First, we define what it means for a delta set Δ to *cover* a given statement A from a given set of states $S = \{s_1, s_2 \dots s_n\}$:

$$\begin{aligned}\text{cover}(\Delta, A, S) \Leftrightarrow &(\forall s \in S, s' \in \overline{DS} : [s, s']_A \Rightarrow (\exists \delta \in \Delta : s' = s \oplus \delta)) \\ &\wedge \ (\forall s \in S, \delta \in \Delta : [s, s \oplus \delta]_A)\end{aligned}$$

I.e., a set of deltas Δ covers a statement A from a set of states S if and only if every state transition achievable from any state in S via A is achievable from the same state via some delta in Δ, and any transition achievable from any $s \in S$ via any delta in Δ is a transition of A.

A delta cover precisely describes all possible executions of a statement from a set of states using a single set of deltas. Intuitively, the existence of a delta cover shows that the given statement does not need to distinguish between the covered states in order to decide how to proceed with execution.

Note that this does not mean that Δ is the collection of all deltas achievable from states in S via A. We can demonstrate this by considering a delete-all statement with $S = \overline{DS}$, and Δ containing a single delta that creates no entities and deletes all entities that exist in any state in $\overline{DS}$. In this particular case, the delta in Δ is different than any delta achievable from any finite $s \in S$ via A, yet it is true that this Δ covers A from S.

We can now define what it means for a statement A to *read* an entity e:

$$\text{reads}(A, e) \Leftrightarrow \exists s \in \overline{DS} : \neg\exists \Delta \subseteq \overline{DS}_\Delta : \text{cover}(\Delta, A, \{s \cup \{e\}, s \setminus \{e\}\})$$

This means that A reads e if and only if there exists a pair of states $s \cup \{e\}$ and $s \setminus \{e\}$ that cannot be covered by any Δ for the statement A. This implies that A's actions are dependent on e's existence in some way, for example if it is deciding whether to delete or not delete some object other than e based on e's existence. I.e., if statement A reads entity e, then in order to describe the behavior of A, we need to specifically refer to e.

Based on this definition, a delete-all statement does not read any entity e because, for any two states $s \cup \{e\}$ and $s \setminus \{e\}$ for any state s, there exists a Δ that covers it: $\Delta = \{\langle\{\}, \{\}, \text{objects of } (s \cup \{e\}), \text{tuples of } (s \cup \{e\})\rangle\}$. Hence, using this definition, we are able to infer that two delete-all statements that are executed back to back are co-executable, although in sequential execution, behavior of the second delete-all statement changes (it becomes a no-op) due to the presence of the first delete-all statement.

We can also define what it means for a statement A to *create* or *delete* an entity similarly:

$$\begin{aligned}\text{creates}(A, e) &\Leftrightarrow \exists s, s' \in \overline{DS} : [s, s']_A \wedge e \notin s \wedge e \in s' \\ \text{deletes}(A, e) &\Leftrightarrow \exists s, s' \in \overline{DS} : [s, s']_A \wedge e \in s \wedge e \notin s'\end{aligned}$$

Recall that, since we are abstracting away the basic types, any update to the data store state consists of creation and deletion of entities (i.e., objects and associations).

2) Coexecutability Condition Definition and Proof: We can now define the coexecutability condition and our main result:

Theorem 1: Given two statements A and B, if the following condition holds:

$$\begin{aligned}\forall s \in \overline{DS}, \forall e \in s : &(\text{reads}(A, e) \Rightarrow \neg\,\text{creates}(B, e) \wedge \neg\,\text{deletes}(B, e)) \\ &\wedge(\text{creates}(A, e) \Rightarrow \neg\,\text{reads}(B, e) \wedge \neg\,\text{deletes}(B, e)) \\ &\wedge(\text{deletes}(A, e) \Rightarrow \neg\,\text{reads}(B, e) \wedge \neg\,\text{creates}(B, e))\end{aligned}$$

then coexecution of A and B is equivalent to their sequential execution (i.e., $\forall s, s' \in \overline{DS} : [s, s']_{A;B} \Leftrightarrow [s, s']_{A|B}$). In other words, A and B are *coexecutable*.

The proof of the above theorem is tedious due to the differences between objects and tuples and how they depend on one another (e.g., deleting an object deletes all associated tuples, and creating a tuple that is associated with a non-existing object is impossible etc.). In order to simplify the proof, without loss of generality, we will outline the proof by focusing only on the creation and deletion of objects.

First, the condition in Theorem 1 implies that no statement can delete an object that can be created by the other. Therefore the deltas from any s via A and B are not conflicting, and coexecution is always defined.

Let us take any two states s and s' and assume that there exists a state s_A such that $[s, s_A]_A$.

Let us consider any object o_c that is created by $s_A \ominus s$. All objects created by $s_A \ominus s$ are not read by B. Therefore, there exists a delta cover Δ that describes all transitions from $s \cup \{o_c\}$ and $s \setminus \{o_c\}$ via B. Since $s_A \ominus s$ is creating o_c we know that $o_c \notin s$, so this delta cover describes all transitions from s and $s \cup \{o_c\}$ via B.

Let us inspect every member of such a delta set Δ. If any $\delta \in \Delta$ creates anything in s then this operation is always redundant for states s and $s \cup \{o_c\}$, so we can remove this operation and still have a delta cover of B over $\{s, s \cup \{o_c\}\}$. We can similarly remove all deletions of all objects outside $s \cup \{o_c\}$ as redundant operations. Since o_c cannot be deleted by B, we know that this trimmed delta cover does not delete anything outside s. From the definition of a delta cover it follows that the resulting trimmed delta cover is, in fact, precisely the set of all deltas achievable from s via B.

Similar reasoning can be followed for any object o_d that is deleted by $s_A \ominus s$. It follows that the set of deltas achievable from s via B covers B over $\{s, s \setminus \{o_d\}\}$.

Because the set of all deltas achievable from s via B covers $\{s, s \cup \{o_c\}\}$, directly from the definition of delta covers (with the prior assumption that an s_A s.t. $[s, s_A]_A$ exists):

$$\begin{aligned}&\forall s, s' \in \overline{DS} \times \overline{DS} : \exists s_A : [s, s_A]_A \Rightarrow \\ &\quad([s \cup \{o_c\}, s']_B \Leftrightarrow \exists s_B \in \overline{DS} : [s, s_B]_B \wedge s' = s_B \cup \{o_c\})\end{aligned}$$

Let us generalize and say that $s_A \ominus s$ creates objects o_{ci} for some $1 \leq i \leq n_c$ and deletes objects o_{di} for some $1 \leq i \leq n_d$. If we were to now enumerate all objects created and deleted by $s_A \ominus s$ one by one and apply the above reasoning to them, the resulting formula would be:

$$\begin{aligned}&\forall s, s' \in \overline{DS} \times \overline{DS} : \exists s_A \in \overline{DS} : [s, s_A]_A \Rightarrow \\ &\quad([s \cup \{o_{c1}, \dots o_{cn_c}\} \setminus \{o_{d1}, \dots o_{dn_d}\}, s']_B \Leftrightarrow \exists s_B \in \overline{DS} : \\ &\quad[s, s_B]_B \wedge s' = s_B \cup \{o_{c1}, \dots o_{cn_c}\} \setminus \{o_{d1}, \dots o_{dn_d}\})\end{aligned}$$

Category	Node	Children
Statement	`Block`	**Statement*
	`Either`	**Block*
	`If`	*Formula*, `Block`, `Block`
	`ObjectSetStmt`	*Object Set*
	`Assign`	`Variable`, *Object Set*
	`Delete`	*Object Set*
	`CreateTuple`	*Object Set*, `Relation`, *Object Set*
	`DeleteTuple`	*Object Set*, `Relation`, *Object Set*
	`ForEach`	`Variable`, *Object Set*, `Block`
Object Set	`Variable`	
	`CreateObjectSet`	`Class`
	`Dereference`	*Object Set*, `Relation`
	`AllOfClass`	`Class`
	`Subset`	*Object Set*
	`OneOf`	*Object Set*
	`Union`	**Object Set*
	`Empty`	
	`DereferenceCreate`	*Object Set*, `Relation`

Fig. 4. Abstract Data Store Statement and Object Set Nodes

Which is equivalent to

$$\forall s, s' \in \overline{DS} \times \overline{DS} : \exists s_A \in \overline{DS} : [s, s_A]_A \Rightarrow$$
$$[s \oplus (s_A \ominus s), s']_B \Leftrightarrow \exists s_B \in \overline{DS} : [s, s_B]_B \wedge s' = s_B \oplus (s_A \ominus s)$$

Because $s_B = s \oplus (s_B \ominus s)$, and applying non-conflicting deltas in sequence is equivalent to applying their union, this formula is equivalent to

$$\forall s, s' \in \overline{DS} \times \overline{DS} : \exists s_A \in \overline{DS} : [s, s_A]_A \Rightarrow$$
$$([s_A, s']_B \Leftrightarrow \exists s_B \in \overline{DS} : [s, s_B]_B \wedge s' = s \oplus (s_B \ominus s) \cup (s_A \ominus s))$$

We can move the s_A quantification and implication ($\exists s_A \in \overline{DS} : [s, s_A]_A \Rightarrow \ldots$) to both sides of the inside equivalence:

$$\forall s, s' \in \overline{DS} \times \overline{DS} :$$
$$(\exists s_A \in \overline{DS} : [s, s_A]_A \wedge [s_A, s']_B) \Leftrightarrow$$
$$(\exists s_A, s_B \in \overline{DS} \times \overline{DS} : [s, s_A]_A \wedge [s, s_B]_B$$
$$\wedge\, s' = s \oplus (s_B \ominus s) \cup (s_A \ominus s))$$

which is the formula for equivalence of sequential execution and coexecution.

V. Syntactic Analysis

In order to keep our verification process fully automatic, we developed a syntactic check that determines, for a given `ForEach` loop, whether we can coexecute the iterations while maintaining the loop semantics. Our syntactic analysis works on an intermediate abstract data store (ADS) language [5], and we automatically extract ADS language specifications from Rails applications using the techniques presented in [5].

The summary of all statements in our ADS language is provided in Figure 4. The ADS language includes constructs for creating and deleting objects (`CreateObjectSet`, `DereferenceCreate`, `Delete`), updating associations (`CreateTuple`, `DeleteTuple`), variables and assignments (`Variable`, `Assign`), loops (`ForEach`), conditional and non-deterministic branches (`If`, `Either`). Most statements use expressions in the form of *Object Sets*. For example, a `Delete` statement takes an Object Set expression and will delete all objects inside this set. We check the coexecutability condition on `ForEach` statements.

The syntactic check is two-fold: 1) we analyze if sequential execution is necessary to uphold variable dependencies, and 2) if iteration operations may overlap as defined in the coexecutability condition (Theorem 1).

```
1  program Analysis
2  var data: AnalysisData;
3
4  function Analyze(loop: ForEach): Boolean
5    data.clearAllData;
6    AnalyzeStatement(loop);
7    for type in DataStoreTypes:
8      operations = data.operationsDoneOn(type);
9      if (operations.hasTwoDifferentOpsWithOneGlobal()) then
10       return False;
11     end;
12   end;
13   return True;
14 end
15
16 procedure AnalyzeStatement(stmt: Statement)
17   case type(stmt) of
18   Block:
19     for subStmt in stmt.subStatements do
20       AnalyzeStatement(subStmt);
21     end;
22   Delete:
23     <objSetType, objSetLocal> = AnalyzeObjset(stmt.objSet)
24     data.markDelete(objSetType, objSetLocal);
25     for relation in objSetType.associations do
26       data.markDelete(relation, False);
27     end;
28   ObjectSetStmt:
29     AnalyzeObjset(stmt.objSet);
30   Assignment:
31     <objSetType, objSetLocal> = AnalyzeObjset(stmt.objSet);
32     stmt.variable.objSetType = objSetType;
33     stmt.variable.objSetLocal = objSetLocal;
34   ForEach:
35     <objSetType, objSetLocal> = AnalyzeObjset(stmt.objSet);
36     data.markRead(objSetType, objSetLocal);
37     stmt.iteratorVariable.objSetType = objSetType;
38     stmt.iteratorVariable.objSetLocal = objSetLocal;
39     AnalyzeStatement(stmt.block, data);
40   ...
41   end;
42 end
43
44 function AnalyzeObjset(objSet: ObjectSet): <Type, Boolean>
45   case type(objSet) of
46   CreateObjsetSet:
47     data.markCreate(objSet.createdType, True);
48     return <objSet.createdType, True>;
49   Variable:
50     return <objSet.objSetType, objSet.objSetLocal>;
51   Dereference:
52     <originType, originLocal> =
53         AnalyzeObjset(objSet.originObjSet);
54     data.markRead(originType, originLocal);
55     data.markRead(objSet.relation, False);
56     return <objSet.targetType, False>;
57   ...
58   end
59 end
```

Fig. 5. Syntactic Analysis Pseudocode

First, to check if coexecution would invalidate variable dependences, we convert the whole action to static single assignment (SSA) form. If, after converting to SSA and removing unnecessary assignments, there exists a Phi function assignment at the beginning of the loop's iteration body (i.e. an iteration reads a variable assigned to by a previous iteration) or at the end of the loop (i.e. the iteration assigns to a variable that is read after the loop terminates), then iterations must be modeled sequentially to preserve variable state.

If the variable dependency check passes, we proceed to check whether the loop is coexecutable. To achieve this, we identify every data store class or relation that is touched by a read, create or delete operation inside the iteration body.

For example, if a `delete` statement deletes a set of objects of class c, we mark that c as well as all c's subclasses have

had a delete operation executed. In addition, since all tuples of deleted objects are deleted as well, we mark all relations of these classes and their supertypes as having had a delete operation executed.

We increase the precision of our analysis by identifying whether operations are executed on iteration-local objects. For example, if an iteration were to create an object of class c and subsequently delete it, then the coexecutability condition would not be violated (since no object created by one iteration would be deleted by another) but the above syntactic check would fail as c would have had both a create and a delete operation executed.

In order to facilitate this we denote whether each read, create or delete operation is done iteration-locally or not. For example, `CreateObjectSet` creates an object iteration-locally as every iteration will create a different object and these created sets will not overlap. Operations such as dereferencing from an object set return a global domain object set even if a local domain was dereferenced, because even if each iteration dereferences from a different object domain, the target object sets may overlap.

Therefore, in order for the syntactic check to pass, there must not exist a domain of objects or tuples that has two of the operations (read, create, delete) executed, where at least one of this operations is not done iteration-locally. The pseudocode for the operation domain analysis is provided in Figure 5.

The `AnalysisData` global variable called `data` (line 2) aggregates information about which domains of objects and tuples are operated on and in what way. It is essentially a key-value structure that maps every class and relation in the data store into a set of *operation entries*, which are pairs $\langle o, l \rangle$ where $o \in \{create, read, delete\}$ and $l \in \{True, False\}$. This structure lists, for each data store class and relation, all the different `create`, `read` and `write` operations executed on entities of that particular class or relation and if these operations were executed iteration-locally (`True`) or not (`False`). This structure is populated by invoking methods `markRead`, `markCreate` and `markDelete` on it, all of which take two arguments: a data store class or relation type, and a boolean denoting whether the operation is iteration-local (e.g. line 24).

The `Analyze` function is the entry point of our algorithm (lines 4-14). It first clears all information from the `data` object (line 5), then proceeds to gather information in the `data` object by invoking the `AnalyzeStatement` on the given loop (line 6). It then iterates over all classes and relations (lines 7-12) and tests whether there exist two operation entries on the same class or relation such that they contain different $\{create, read, delete\}$ types where at least one of them is executed on a global domain. If such a pair of operation entries is found, the coexecutability check fails and the function returns `False` (line 10). Otherwise, it returns `True` (line 13).

The `AnalyzeStatement` procedure takes a statement as an argument and its purpose is to populate the `data` object with information about which operations are executed on which domain by that statement. For example, a `Delete` statement (lines 22-27) invokes the `AnalyzeObjset` method to acquire domain information about the object set to be deleted (line 23), then marks this domain as deleted (line 24). Since the `delete` statement also deletes all tuples of the deleted objects, all relations around the object set's type are iterated over (lines 25-27) and are marked deleted globally (line 26). The tuples are always deleted on a global domain because, even if the deletion is on a local domain, these tuples may relate to some other iteration's local domain.

The `Assignment` statement (lines 30-33) does not add any entry to the `data object`, and instead stores the domain of the assigned object set in the variable. These values will later be extracted when the variable is referred to in lines 49-50.

The `AnalyzeObjset` function (lines 44-59) is invoked with an object set argument and it returns the domain of the objects inside the object set in form of a $\langle Type, Boolean \rangle$ pair. In addition, object sets may populate the `data` object themselves. For example, the `CreateObjectSet` object set (lines 46-49) creates a new object and returns a singleton set containing it. For each such object set, we mark that object's class with the create operation (line 47) in an iteration-local domain because this object set will contain a different object for each iteration.

The `Dereference` object set (lines 51-56) takes another object set, referred to as the origin object set, and a relation type. It contains all the objects that can be reached from the origin object set via at least one tuple of the given relation type. As such, the origin object set is read in the domain supplied by it (lines 52-54), and the relation type is read on the global domain (line 54). Finally, the returned domain of this very `Dereference` object set is equal to the target type of the relation and is always global (line 56).

VI. Experiments

We implemented the analysis and verification techniques presented in this paper in our data model verification tool which is available at http://bocete.github.io/adsl/.

In order to evaluate the effect of coexecution on the verification process, we implemented two ways to model `ForEach` loops (as sequentially executed iterations, and as coexecuted iterations). We looked at the top 20 most popular (most starred) Rails applications hosted on Github [16] for real-world examples of loops. Four of them do not use ActiveRecord or a relational database. Two of them are not web applications per se, but rather web application templates, and one has an unorthodox architecture that was not compatible with the data model extraction component of our tool. Out of the remaining 13, 6 of them had no loops in their actions. We found a total of 38 loops in actions of the remaining 7 applications: 5 in Discourse [7], 9 in FatFreeCRM [9], 5 in Tracks [30], 4 in Lobsters [25], 5 in SprintApp [29], 8 in Redmine [27], and 1 in Kandan [21]. Our analysis determined that all these loops were coexecutable.

Interestingly, 12 of the 38 loops we extracted had empty loop bodies. This is due to the fact that the abstract data store model we extract abstracts away the fields with basic types. Hence, the loops that do not modify the state of the data store as far as the set of objects and associations are

concerned (but might change the value of basic type fields of some objects) result in empty loop bodies. Note that, since the invariants we verify are on sets of objects and their associations, the automated abstraction performed by our tool during abstract data store extraction, in effect, is demonstrating that these loops preserve all invariants. However, during our experiments, we found out that the FOL theorem prover would occasionally timeout even for loops with an empty body when the sequential semantics is used. This demonstrates the inherent complexity of reasoning about the sequential loop model, even without the complexity of reasoning about the statements in the loop body.

The remaining loops we extracted contain various program structures such as branches, object and association creation and deletion as well as loop nesting.

We manually wrote application specific invariants for each application to be used in our experiments. Given an action and an invariant, our verification tool generates a set of FOL formulas that correspond to axioms, and a formula that corresponds to a conjecture. If the axioms imply the conjecture, then this proves that the action preserves the invariant.

We used the Spass [32] tool as the FOL theorem prover in our experiments. Spass takes the set of FOL axioms and the conjecture generated by our tool and attempts to prove that the axioms imply the conjecture. More precisely, Spass conjoins the negation of the conjecture with the axioms and starts deducing formulae from this set of axioms. If a contradiction is ever reached, the conjecture is deemed to hold. If the entire space of deducible formulae is exhausted, the conjecture is deemed to not imply from the axioms. Finally, since FOL is undecidable and the space of deducible formulae may be infinite, deduction may never terminate. We stop verification after 5 minutes, at which point we deem the result as inconclusive.

Spass guides the formula space exploration using heuristics that can be fine tuned by the user. We used two heuristics: one with the *Sorts* option on, and the other with that option off. The Sorts option allows Spass to make decisions based on soft sorts [31]. By turning the Sorts option on and off and looking at the deduction logs of Spass we noticed that the order of deduction Spass takes changes significantly, and furthermore, in our previous work [5] we found that, often, one of these heuristics terminates when the other one does not. Therefore, running Spass with the Sorts option on and off gives us very different heuristics for comparison of coexecution and sequential execution.

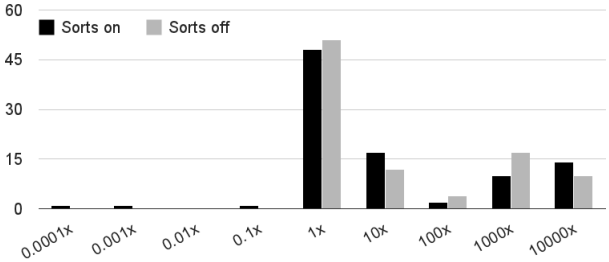
Normally, when we encode an action/invariant pair in FOL, the loop semantics are encoded in the axioms section with an invariant being the conjecture. In order to isolate the effect of the axioms on the overall deduction process, we also verified all actions using the conjecture *false*. This conjecture often gives us the worst case performance for a set of axioms. Because Spass attempts to deduce a contradiction from the axioms and negated conjecture, it negates the conjecture to *true* and hence needs to explore the entire space of deducible formulae to reach a contradiction that does not exist. If Spass terminates with the *false* conjecture then we can reasonably

Application	KLOC	# of Model Classes	# of Loops	# Avg Loop Body Size
Discourse	308.77	116	5	1.4
FatFreeCRM	38.79	34	9	1.0
Tracks	28.72	39	5	2.0
Lobsters	7.99	17	4	3.0
SprintApp	7.89	19	5	3.2
Redmine	153.43	67	9	1.1
Kandan	6.10	11	1	0

(a) Application Statistics

Loop Model	Heuristic	# of Timeouts / Total	Avg Time (seconds)
Sequential	Sorts on	66/94 (70.2%)	216.4
	Sorts off	56/94 (60.2%)	186.0
Coexecution	Sorts on	27/94 (29.3%)	94.0
	Sorts off	18/94 (19.8%)	68.5

(b) Verification statistics



(c) Number of action/invariants per performance gain factor

Fig. 6. Application and Verification Statistics

expect that it will terminate with other invariants as long as they do not add significant complexity to the verified theorem.

We also included 4 actions that we manually created to explore how the theorem prover handles coexecution vs sequential execution of nested loops and branches in iterations.

In total, we had 94 action/conjecture pairs. We translated each one of those to two FOL theorems, one using coexecution and the other using sequential execution to model loops. We sent each one of these theorems to two instances of Spass with different heuristic settings, resulting in 376 verification tasks. All verification experiments were executed on a computer with an Intel Core i5-2400S processor and 32GB RAM, running 64bit Linux. Memory consumption never exceeded 200Mb.

We specifically looked at how coexecution fared as opposed to sequential execution. With the sequential execution model, out of 188 verification tasks 122 of them timed out (64.89%). With the coexecution model, only 45 tasks out of 188 timed out (24.19%). The summary of our results can be seen on Figure 6. Figure 6(a) summarizes information on the applications we used for evaluation. Figure 6(b) summarizes the number of timeouts and average verification time over loop interpretation (sequential vs coexecution) and heuristic.

Figure 6(c) summarizes the performance effects of coexecution as opposed to sequential execution. We had 188 cases in which to compare coexecution and sequential execution under identical action/invariants and theorem prover heuristics. In 86 cases (45%, columns labeled *10x* and up), coexecution improved verification times by at least an order of magnitude. Among them, in 24 cases (13%) the theorem prover reached a conclusive answer instantly using coexecution and could not deduce a conclusive answer at all with sequential execution (column labeled *10000x*). Coexecution yielded no

improvement in a total of 99 tasks (52%, column *1x*). In these cases either both loop models resulted in a timeout or both methods produced results instantly. In three cases, coexecution produced worse results than the sequential model. This is not surprising since, as we mentioned above, the proof search implementation of the theorem prover relies on several heuristics which influence its performance.

In total, we found that coexecution reduced the timeout rate from 65% to 24% (almost threefold), made verification at least an order of magnitude faster 45% of the time, with 13% of cases terminating quickly as opposed to not terminating at all. We conclude that, overall, coexecution allows for significantly faster verification and significantly decreases the chance of verification never terminating.

VII. Related Work

The contributions of this paper are motivated by our previous work on data model verification [5]. In our previous work we presented an imperative language for web application data modeling called abstract data store language (ADS), a method for automated extraction of this language from a given Rails web application, as well as the process of translating ADS verification tasks to FOL. In contrast to our earlier work, this paper focuses on effective verification of actions that contain loops, which is one of the key difficulties in the FOL-based verification of data models. Using coexecution to model loops is applicable to program verification in general, beyond the scope of our previous work.

Verification of software using theorem provers has been explored before in projects such as Boogie [3], Dafny [24] and ESC Java [12]. These projects focus on languages such as C, C#, and Java, and typically require user guidance in the form of explicit pre- and post-conditions, explicit data structure constraints, and loop invariants. While loop invariants may be inferred for certain loops, they are ultimately required to reason about the loops. Our method does not require loop invariants, and uses static analysis to automatically optimize the loop translation to FOL. While low level languages such as C and Java present different challenges then our high level language, we believe that modeling loops via coexecution is applicable and would be beneficial for the verification of loops in low level languages as well.

The lack of support for precise reasoning about programming language constructs in theorem provers has been noticed and addressed before [6]. Specifically, [6] discusses this problem with regard to ANSI-C basic types and operations, bit-vectors and structures, pointers and pointer arithmetic. They address this problem by devising a theorem prover that supports all these elementary operations. These improvements do not improve on the basic problems with loop verification, and the tools that use Simplify still require loop invariants [12].

Alloy [19], [20] is a formal language for specifying object oriented data models and their properties. Alloy Analyzer is used to verify properties of Alloy specifications. Alloy Analyzer uses SAT-based bounded verification techniques as opposed to the FOL based unbounded verification technique we use. DynAlloy is an extension of Alloy that supports dynamic behavior [13], [14] by translating dynamic specifications onto Alloy. While they talk about *actions* in their work, those actions do not correspond to actions in web applications. Instead, they are more similar to statements in programming languages [15]. Their work has focused on verification of data structures, not behaviors in data models of web applications.

An interesting parallel can be drawn between coexecution of loop iterations and snapshot isolation in the domain of databases [4], [10]. The coexecutability problem is similar to the problem of equivalence of serializability and snapshot isolation. However, we see no parallel between our delta union and the delta apply operations and snapshot isolation notions such as *first-committer-wins*, transactions aborting or committing based on conflicts etc. Our purpose is verification viability, not scalability or optimization of transactions.

There exists a long body of work focusing on operation commutativity with applications such as automatically parallelizing data structures [22] and computation [18], [28]. Automatic loop parallelization has been researched for decades [1], [2], [17]. This prior research acknowledges loop dependencies as problematic for parallelization, and the potential for performance increase if no such dependencies exist. While we are also avoiding loop dependencies, our purpose is not optimization or making execution scalable, but making verification more feasible in practice. Coexecution is a theoretical concept that is not executable in actual hardware. Furthermore, there exist parallelizable loops that are not coexecutable.

Semantic properties of operations have been used for the purposes of simplifying verification [8]. This is similar to our approach at a high level. However, we do static analysis of a particular condition that allows us to use a completely alternate definition of a loop, whereas [8] iteratively abstracts and subsequently reduces the model in order to infer and enhance atomicity rules without altering the validity of the given invariants. Their problem, domain of application, goal and solution are fundamentally different.

VIII. Conclusion and Future Work

In this paper we defined coexecution, a technique for modeling loop iterations that bypasses the need to model inter-iteration dependence, and the coexecutability condition which tests whether, when given a loop, coexecution of its iterations is equivalent to its sequential execution. We developed a static program analysis technique that tests the coexecutability condition on a given loop. Finally, we demonstrated that modeling loops using coexecution significantly improves verification viability and performance.

We believe that coexecutability can be useful in other contexts and we plan to investigate its application to other verification problems. We also plan to continue and extend our work on automated verification of data stores. For example, basic data types are completely abstracted away in our data model specifications. Support for basic data types coupled with an automated abstraction mechanism could enable us to check a richer set of properties on data models in the future.

REFERENCES

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (SIGPLAN 1988)*, PLDI '88, pages 308–317, New York, NY, USA, 1988. ACM.

[2] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE (1993)*, 81(2):211–243, 1993.

[3] 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.

[4] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD 1995)*, pages 1–10, 1995.

[5] I. Bocic 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.

[6] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of the 17th International Conference on Computer Aided Verification, (CAV 2005)*, pages 296–300, 2005.

[7] Discourse, Mar. 2014. www.discourse.org.

[8] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 2–15, 2009.

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

[10] A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

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

[12] 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.

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

[14] M. F. Frias, C. L. Pombo, J. P. Galeotti, and N. Aguirre. Efficient analysis of dynalloy specifications. *ACM Transactions on Software Enginnering Methodology*, 17(1), 2007.

[15] J. P. Galeotti and M. F. Frias. 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*, pages 249–260, 2006.

[16] GitHub, Sept. 2014. http://github.com.

[17] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[18] O. H. Ibarra, P. C. Diniz, and M. C. Rinard. On the complexity of commutativity analysis. *International Journal of Foundation of Computer Science*, 8(1):81–94, 1997.

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

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

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

[22] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, pages 528–541, 2011.

[23] 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.

[24] 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.

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

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

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

[28] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS 1997)*, 19(6):942–991, 1997.

[29] macfanatic/SprintApp, Sept. 2014. https://github.com/macfanatic/SprintApp.

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

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

[32] 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.