Coexecutability: How To Automatically Verify Loops

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

***Abstract*—Verification of web applications is a very important problem, and verifying loops is necessary to achieve that goal. However, loop verification is a long studied and very difficult problem. We find that interdependence of iterations is a major cause of this difficulty. We present coexecution - a way to model a loop that avoids the problem of iteration interdependence. We introduce the coexecutability condition that implies that coexecution is a correct model. Through experiments, we demonstrate that coexecution reduces the number of inconclusive verification results by three times, and in 43% of cases increases performance of verification by at least an order of magnitude.**

Keywords*—*Verification, Loops

I. Introduction

Web applications are integral to the functioning of the modern society. As such, their correctness is of fundamental importance. In our recent work [3], we focused on automated verification of data store properties in web applications. For example, in an online forum application involving `User`s and `Post`s, our approach could be used to automatically answer questions such as "Does every `Post` have an associated `User`?". In Ruby on Rails [4], a `delete_user` action could be implemented as shown in Figure 1.

```
class UsersController
  def destroy_user
    user = User.find params[:user_id]
    user.posts.each do |post|
      post.destroy
    end
    user.destroy
  end
end
```

Fig. 1. Rails Code for User Deletion

Our tool analyzes the actions of a given web application using automated theorem proving techniques to deduce whether a given property is preserved by all actions.

Verification of loops is a long studied, difficult and generally undecidable problem [2], [1], often requiring manual intervention in form of loop invariants. We found that the main problem with verification of loops stems from automated reasoning about of how iterations affect each other. In general, the sequence of iterations is of an arbitrary length, with any iteration being potentially affected by all previous iterations. When reasoning about a loop's behavior, an automated theorem prover would enumerate iterations one by one to deduce all the possible results of a loop, producing increasingly more complex formulas, and may never terminate.

In this paper we identify a special class of loops for which modeling iteration interdependence is not necessary. We define a concept called *coexecution*, which models loop iteration executions in a way that avoids iteration inter-dependability. We define a condition under which coexecution is equivalent to sequential execution. Through experiments, we show that coexecution significantly improves the viability and performance of loop verification.

II. Formalization

For brevity and simplicity, we refer to the elements of a data store as *entities* without going into specifics of their nature. Think of them as objects that represent the database, associated to one another as defined by the schema. For similar reasons, we will assume that these entities contain no mutable data, and can only be created or deleted. Modifications could be implemented as deletion followed by creation of a similar entity populated with the updated data.

A data store *state* is a set of entities that exist in some point in time, for example, a set of `User` objects and their associated `Post`s. A statement serves to update the state. We define a statement S as a set of state pairs such that executing the statement S from state s may result in state s' if and only if $(s, s') \in S$. For example, a statement that creates an entity e can be defined as a set of state pairs (s, s') s.t. $e \notin s$, $e \in s'$, and $\forall e' : e \neq e' \rightarrow (e' \in s \leftrightarrow e' \in s')$.

At a high level a loop is defined as a sequential execution of k iterations, where k is the size of the set being iterated on. While all iterations execute the same loop body, in our formal model, we treat each iteration as a unique statement. Hence, a loop is modeled as the sequential execution of iteration statements $S_1, S_2, \ldots S_k$ that migrates state s_0 to s_k if and only if:

$$\exists s_1, \ldots s_{k-1} : (s_0, s_1) \in S_1 \wedge \cdots \wedge (s_{k-1}, s_k) \in S_k$$

Our key problem comes from this definition. To describe possible migrations from s_0 to s_k, the theorem prover needs to deduce all possible s_1s that are reachable from s_0, then s_2s from the s_1s etc. The complexity of each subsequent state increases and, since k can often be any integer, this process may never terminate.

III. Solution

When we manually investigated loops in web applications, we found out that, typically, loop iterations do not affect each other. Therefore, modeling the rules of how iterations affect each other is not necessary, and is in fact not desirable because this dependence is the major problem of verification feasibility. For example, the loop iterations in Figure 1 are not inter-dependent.

We introduce coexecution, a way to model the composition of multiple statements that are not inter-dependent. Coexecution entails identifying the effects of each iteration's execution as if this iteration were executed in isolation from all others, and combining these effects into one major operation that we can use to model the loop.

To formally express this process, we first need to express the effects of a statement's execution. We define a *delta* of states s and s' (denoted as $(s' \ominus s)$) to be a structure (O_c, O_d) where O_c is the set of entities that exist in s' but not in s (created entities), and O_d is the set of entities that exist s but not in s' (deleted entities).

Next, we need a way to combine multiple deltas into one delta that encompasses all of them. We combine deltas $\delta_1 = (O_{c1}, O_{d1})$ and $\delta_2 = (O_{c2}, O_{d2})$ by using the *delta union* ($\cup$) operator, defined as $\delta_1 \cup \delta_2 = (O_{c1} \cup O_{c2}, O_{d1} \cup O_{d2})$. Note that, in general, the same entity may be both in the create and the delete set of the result of a delta union. We call a delta *conflicting* if its delete and create sets are not exclusive.

Finally, we need a way formalize the execution of a delta. Given a state s and a non-conflicting delta $\delta = (O_c, O_d)$, we can *apply* δ to s using the operator $\oplus$: the result of that operation will be a state that contains all elements of $(s \cup O_c) \setminus O_d$. The apply delta operation is undefined for conflicting deltas.

We can finally define coexecution. The coexecution of statements $S_1, \dots S_k$ migrates between states s and s' iff:

$$\exists s_1 \dots s_k : (s, s_1) \in S_1 \wedge \dots \wedge (s, s_k) \in S_k \wedge \\ s' = s \oplus (s_1 \ominus s) \cup \dots \cup (s_k \ominus s)$$

A. The Coexecutability Condition

In general, the result of coexecution of statements is different from the result of sequential execution. For example, in sequential execution, a statement may undo some of the previous statement's work, or may read the modifications done by a previous statement and behave differently because of that, even if never undoing previous work. These possibilities would not be captured by the coexecution approach, since coexecution models the execution of all statements in isolation. Therefore, we need a condition under which sequential execution and coexecution are equivalent - the coexecutability condition.

It is intuitive that these two models of execution should be equivalent if no statement reads what another creates or deletes, and if no statement creates what another has deleted or vice versa. However, we defined statements as arbitrary migrations between states, and so the very notion of *reading* an entity is undefined.

Intuitively, a statement reads an entity if the statement's semantic or intent changes depending on the presence of the read entity. However, this is a problematic intuition since, for example, creating or deleting an entity is possible if and only if that entity does not or does exist, respectfully. That means that any modification of the data implies reading said data, which is a very limiting factor. Consider a loop in which every iteration deletes all data. Coexecution is equivalent to the sequential execution for this loop, yet if all iterations delete all data they also read all data, implying that the coexecutability condition fails. This intuition is too coarse.

We define reading in a more flexible way. Let's introduce the concept of a *delta cover* Δ of a statement S and a set of states α as a set of deltas such that all executions of S from any state in α is expressed by at least one delta from Δ, and that applying any delta from Δ to any state in α is accepted by the statement S. Defined like this, a delta cover exists for a statement S for these states only if S has a certain semantic (which is the set of deltas) that is not different between the given states. In words, it does not differentiate them.

Using the concept of delta covers, we can define what it means for a statement to read an entity e. A statement reads an entity e if and only if there exists a pair of states such that the only difference between them is the existence of e, and there exists no delta cover of S over these two states.

Let us reconsider the loop that deletes all data in every iteration, within this new definition of reading. The delete all statement does not read any entity because there exists a delta cover over any pair of states as defined above: this delta cover contains a single delta that has an empty create set, and all possible entities in the delete set. This makes sense because a delete all statement is not affected by the existence of any entity, it always does the same thing. In this loop, because all iterations delete the same set of entities without undoing or being guided by other iterations, this loop is coexecutable.

Note that coexecution, while seeming similar to parallelization, is fundamentally different. Consider a loop that, on each iteration, deletes all entities and subsequently creates an entity in one atomic step. If iterations were parallelized, the end result of the loop will always have exactly one entity in the state. If these iterations were coexecuted, the end result will have as many entities as there were iterations. This loop is not coexecutable, but it demonstrates the difference between parallelization and coexecution.

IV. Experiments and Conclusion

We measured and compared the performance of verification on four open source Ruby on Rails applications when using coexecution vs sequential models of loops. We run our experiments under different heuristics in order to demonstrate that coexecution is generally easier to automatically reason about.

Our results show that, by using coexecution, the number of inconclusive results involving loop verification is reduced more than threefold - from 62% down to 17%. Furthermore, the performance of verification is largely improved. In 43% of cases, coexecution yields an improvement of at least an order of magnitude. In 22% cases, coexecution provided an improvement of four orders of magnitude.

Our results demonstrate that automated verification of web application actions is feasible in most cases, even when they contain loops.

References

[1] T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, pages 260–264, 2001.

[2] 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.

[3] 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.

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