Verification of ORM-based Controllers by Summary Inference

Geetam Chawla
Indian Institute of Science
Bangalore, India
geetam.chawla@gmail.com

Navneet Aman
Indian Institute of Science
Bangalore, India
navneetankur@gmail.com

Raghavan Komondoor
Indian Institute of Science
Bangalore, India
raghavan@iisc.ac.in

Ashish Bokil
Indian Institute of Science
Bangalore, India
ashishsb@iisc.ac.in

Nilesh Kharat
Indian Institute of Science
Bangalore, India
nileshramesh@iisc.ac.in

ABSTRACT

In this work we describe a novel approach for modeling, analysis and verification of database-accessing applications that use the ORM (Object Relational Mapping) paradigm. Rather than directly analyze ORM code to check specific properties, our approach infers a general-purpose relational algebra summary of each controller in the application. This summary can then be fed into any off-the-shelf relational algebra solver to check for properties or specifications given by a developer. The summaries can also aid program understanding, and may have other applications. We have implemented our approach as a prototype tool that works for 'Spring' based MVC applications. A preliminary evaluation reveals that the approach is efficient, and gives good results while checking a set of properties given by human subjects.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Software functional properties**; • **Information systems → Web applications**.

KEYWORDS

program analysis, database applications, relational algebra

ACM Reference Format:
Geetam Chawla, Navneet Aman, Raghavan Komondoor, Ashish Bokil, and Nilesh Kharat. 2022. Verification of ORM-based Controllers by Summary Inference. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510148

1 INTRODUCTION

MVC (Model-View-Controller) *frameworks* are regularly used to develop web applications and *RESTful* services [8]. An MVC application consists primarily of a set of *controllers*, each of which receives requests directed to a specific URL. Much of the core logic in these controllers tends to be focused on fetching or updating data in databases. Therefore, MVC frameworks commonly include ORM (Object Relational Mapping) APIs, which make database access intuitive for programmers. The ORM idea is basically to let the developer use imperative constructs such as loops, and access or update a database table as if it were an in-memory collection of *entities*, where entities are in-memory objects representing the tuples in the database. ORM is frequently preferred by developers over embedded SQL (via ODBC APIs) as it avoids impedance mismatch between SQL and imperative code as well as type- and schema-related errors.

While the imperative flavor of ORM code is intuitive to many developers, mechanically checking properties of imperative code is more challenging than mechanically reasoning about SQL, which is declarative in nature. We propose to bridge this gap by proposing in this paper a novel approach that infers a declarative summary, in *relational-algebra* form, of the database updates performed by a controller and of the *model attributes* that it may return.

```
1 @PostMapping("/setDefaultPayment")
2 public String setDefaultPayment(Long defPayId, Model
      model, Principal principal){
3   User user = userService.findByUsername(principal.
      getName());
4   userService.setUserDefaultPayment(defPayId,user);
5   ...
6 }
7
8 UserService :: public void setUserDefaultPayment(Long
      defPayId, User user) {
9   List<UserPayment> uPList = uPRepo.findAll();
10  for(UserPayment uPIter: uPList){
11    if(uPIter.getId()==defPayId){
12      uPIter.setDefaultPayment(true);
13      uPRepo.save(uPIter);
14    }else{
15      uPIter.setDefaultPayment(false);
16      uPRepo.save(uPIter);
17    }
18  }
19 }
```

Figure 1: Example controller

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510148

1.1 Motivating Example

Figure 1 depicts a controller[1] named `setDefaultPayment` from an open-source book-store application [9] in Spring. The controller in turn calls the method `setUserDefaultPayment`. This method first retrieves all "user payment" entities from a table named `userPayment` in Line 9. The *repository* variable 'uPRepo' corresponds to the table `userPayment`. Repository variables are Spring's interfaces to database tables; each repository variable implicitly provides method calls to query the underlying table using the fields of the table and to save tuples into the underlying table.

In the example, each 'UserPayment' entity represents a payment method (e.g., the information about a credit card). Each of these entities contains a field 'id' (which is the primary key), a boolean field 'defaultPayment', which encodes whether this payment method is the "default" payment method or not for its owning user, and a field 'type', which encodes whether this payment method is a credit card, or a debit card, etc.

In the loop in Lines 10-18, each 'UserPayment' entity's 'defaultPayment' field is set to *true* or *false* depending on whether this entity's 'id' field is equal or not to the given argument 'defPayId'. These updated entities are also saved back into the table `userPayment`.

A developer might want to check if the controller in Figure 1 satisfies certain properties. Consider the following example, which we denote as Property (1): Does the controller set to *true* (resp. *false*) the 'defaultPayment' field of a 'UserPayment' entity only if the entity's 'id' field is equal (resp. not equal) to the argument 'defPayId'? This property is indeed satisfied by the code.

Automated checking of properties such as the ones above would enable the production of reliable software, and this is the problem we address in this paper. Automated property checking for ORM controllers is a challenging problem, for a few different reasons. The first is that in-memory collections are used to store specific subsets of database tables, and these subsets typically need to be characterized with good precision by analysis techniques to find property violations. Secondly, ORM code frequently uses imperative loops to iterate over databases or collections. Generally, loops are known to be challenging for automated property-checking approaches.

1.2 Our approach

In this paper we propose a static analysis approach to infer summaries in relational algebra form from ORM controllers. A summary of a controller maps each database table that is updated in a given controller and each return value from the controller to a relational algebra expression. For the controller 'setDefaultPayment' in Figure 1, in its inferred summary, the expression for the table `userPayment` (which is mapped to the repository variable 'uPRepo') would be as follows.

$$uPRepo \mapsto \Pi_{id,true,type}(\sigma_{id=defPayId}(uPRepo)) \cup \Pi_{id,false,type}(\sigma_{id\neq defPayId}(uPRepo)) \quad (1)$$

Note, the occurrence of 'uPRepo' within the relational algebra expression to the right of the "↦" refers to the incoming contents of the table `userPayment` when the controller is invoked, while the occurrence to the left of '↦' denotes the updated contents when the controller finishes execution. Also, throughout this paper we use a generalized form of the projection operator 'Π' that allows any tuple of expressions in its subscript, and not just a tuple of field names.

```
for (⟨itr⟩ : ⟨coll1⟩) {
  if (⟨cond⟩)
    ⟨coll2⟩.save(⟨tuple1⟩);
  else
    ⟨coll2⟩.save(⟨tuple2⟩);
}
```

⇒

$$\langle coll_2\rangle \mapsto \langle coll_2\rangle \text{ - } \langle coll_1\rangle \cup \Pi_{\langle tuple_1\rangle}(\sigma_{\langle cond\rangle}(\langle coll_1\rangle)) \cup \Pi_{\langle tuple_2\rangle}(\sigma_{\neg\langle cond\rangle}(\langle coll_1\rangle))$$

Figure 2: Code pattern based rewrite rule

Say we want to check Property (1) mentioned in Section 1.1. The developer may specify this property as:

$$\sigma_{id\neq defPayId \,\wedge\, defaultPayment\neq false}(uPRepo) = \emptyset \quad (2)$$

Now, an off-the-shelf relational algebra solver can be used to check that the inferred summary shown in Equation (1) logically implies the specification shown above[2], and hence declare Property (1) as holding.

Summarizing loops (such as the one in Figure 1) is similar to inferring *loop invariants*. This is known to be a challenging problem in program analysis, and would be especially so when loops iterate over collections, copy entities from one collection to another under some condition, etc. To circumvent this difficulty, we propose an efficient pattern-based rewriting technique to infer summaries of loops.

A pattern-based rewrite rule that suffices for our example in Figure 1 is depicted in Figure 2. The names within angle brackets are meta-variables, which match actual expressions in the code. If a rewrite rule's LHS (left hand side) matches a loop, then, intuitively, the summary is obtained by instantiating the RHS (right hand side) pattern. That is, the meta-variables in the RHS are replaced with their matching expressions as obtained from the LHS.

When the LHS of the pattern shown in Figure 2 is matched with the loop in Figure 1, the meta-variable ⟨*itr*⟩ matches the iterator variable 'uPIter', ⟨$coll_1$⟩ matches the collection 'uPList', and so on. After 'uPList' is determined to be equal to 'uPRepo' (using Line 9 in the code), the instantiated RHS becomes equal to the summary shown in Equation (1).

Notwithstanding the simplified intuition mentioned above, the rewriting process is not entirely syntactic or straightforward. For instance, even though ⟨$tuple_1$⟩ syntactically matches 'uPIter' in Line 13 of the code, ⟨$tuple_1$⟩ is replaced in the inferred summary with "*id, true, type*", rather than with any references to 'uPIter'. The transformations referred to above are performed by employing static analysis to infer the values stored in variables. These transformations are necessary because the inferred summary should refer to only the incoming database tables and arguments, and not to the values of local variables or iterators.

[1]The code shown is simplified in minor ways for ease of presentation. Also, while we refer to each request-handling method as a "controller", the general terminology is to refer to a class that may contain several request handling methods as a controller.

[2]This can be done by replacing the 'uPRepo' in Equation (2) with the right-hand-side of Equation (1) and then checking the validity of Equation (2).

1.3 Contributions

Controller summarization. The primary contribution of this paper is a novel approach for summarization of ORM controllers in the form of relational algebra. Our approach is the first one to the best of our knowledge that uses static analysis to infer functional summaries for Java controllers.

Pattern based rewriting. As part of our approach, we introduce a novel, efficient and effective pattern-based rewriting mechanism to summarize a variety of ORM loops.

A related approach that also addresses the problem of summarization of ORM code is by Bocić and Bultan [2]. Their approach attempts to infer loop-invariants for ORM code without any patterns, but is guaranteed to terminate only on certain classes of loops, and abstracts away scalar operations and conditionals.

Reasoning on traces. We introduce an inductive and efficient technique to verify properties of *all possible traces* in a MVC application that pass through specified controllers.

Prototype tool. We describe a prototype implementation of our approach, which is a tool called *ORMInfer*. ORMInfer infers a summary for a given controller method. While our approach conceptually applies to any ORM mechanism, our implementation targets *Spring* [24], which is the most popular MVC framework for Java, and the third most popular backend development framework overall [21]. ORMInfer includes a DSL (domain specific language) to specify pattern-based rewrite rules. Our core implementation infers a relational algebra summary in a solver-independent form, while a postpass translates the summary into the widely used Alloy Language [18] in order to be further checked by the Alloy solver. We have also implemented our trace-property checking approach mentioned above as a tool *MultiORM*, and this tool is dependent on ORMInfer to obtain summaries of the individual controllers that the traces go through.

Evaluation. We applied our tool on six open-source Spring benchmarks, and used the inferred summaries to check properties provided by a set of volunteer graduate students. Our approach identified correct results (pass/fail) on about 76% of the properties, and also demonstrated itself as being very efficient.

The rest of this paper is structured as follows. Section 2 describes our core contribution of summarizing a controller. Section 3 describes the trace-checking extension mentioned above. Section 4 gives an overview of our prototype implementation, while Section 5 presents our empirical evaluation. Section 6 discusses related work, while Section 7 concludes the paper.

2 OUR APPROACH FOR CONTROLLER SUMMARIZATION

In this section we present the core of our approach, which is a syntax-directed technique to infer a relational algebra summary for a given controller.

2.1 Flattening

Inferring summaries for updated repositories and return values requires, as an intermediate step, summaries for local variables as well as heap objects. Heap objects could potentially be modeled using *symbolic objects* provided by points-to analysis. However, points-to analysis is in general expensive, and can also reduce precision by over-approximating information on which variables point to which objects. In bug-detection settings like ours, high precision is desired in order to minimize false positives.

Therefore, we adopt a *flattening* based approach that does not use points-to or alias analysis. The idea intuitively is to model the heap using a set of variables, whose types are primitives or collections but not object references. For instance, if a variable `v1` is an object reference, we replace `v1` with a set of *access paths* of the form `v1.f`, `v1.g`, etc., based on the fields declared in `v1`'s type. If `v1.f` is itself an object reference, we further replace it with a set `v1.f.k`, `v1.f.l`, and so on. We do this at all depths, and retain a variable without further expansion only if its last field is a primitive or a Java collection. This process can go into non-termination in the presence of cyclic references. Therefore, to enforce termination, we impose a length bound on access paths, and throw away any access path that ends in an object reference and that already has as many fields as the bound permits.

After the flattening mentioned above, we effectively treat each access path as if it were a single (non object reference) variable. We start referring to the access paths simply as variables from here on, and correspondingly, use underscores instead of dots in their representations.

Our next step is to replace each original assignment statement with a set of statements that make use of the (flattened) variables obtained above. For instance, a statement of the form "`v1 = v2`" would be replaced with the set (actually, sequence) of statements "*v1_g = v2_g*; *v1_f_k = v2_f_k*; *v1_f_l = v2_f_l*; . . .", the statement "`v1.f = v3`" would be replaced with "*v1_f_k = v3_k*; *v1_f_l = v3_l*; . . .", and so on.

The flattening approach mentioned above can potentially give rise to incorrect summaries in the presence of arbitrary aliasing between variables or access paths. However, it is our observation that real-life Java controllers are very idiomatic, and do not normally setup aliasing between access paths or use the heap in rich ways.

2.2 Summary inference for simple statements

The flattening mentioned above is done as a pre-pass. After the flattening, the rest of the approach is syntax-directed. It essentially performs a bottom-up traversal of the *Abstract Syntax Tree* (AST) of the controller's code, and generates the summary of each subtree using the already generated summaries of the immediately nested subtrees. In this part of the paper we discuss how summaries are inferred for non-looping code fragments.

Figure 3 gives a set of rules for the process described above, one rule per kind of statement. The notation "$S \vdash e$" means that e is the inferred summary of statement S, where e is a mapping from access-paths to relational algebra expressions. The subroutine *mkcond* translates its argument syntactic condition into a condition in the syntax of relational algebra. *Type*(v) returns the name of the table corresponding to the declared type of v (which is an entity class), while *TypeElement*(v) returns the name of the table corresponding to the entities declared to be stored in v provided v is a collection.

Figure 4 illustrates the bottom-up summarization for our running example in Figure 1. Since we are ignoring method calls at this point,

ASSIGN: $\texttt{v1 := v2} \vdash \{(v1, v2)\}$

SEQUENCE:

$$\frac{S1 \vdash V_L \qquad S2 \vdash V_R}{\texttt{S1; S2} \vdash \{(k,v) \mid v = V_R(k)[V_L(g_1)/g_1].....[V_L(g_n)/g_n] \\ \textit{if } k \in \textit{domain}(V_R), \\ \textit{n is the number of leaves in } V_R(k) \\ g_1, g_2 ... g_n \textit{ are leaves in } V_R(k) \\ v = V_L(k), \textit{ if } k \in V_L\}}$$

IF-THEN-ELSE:

$$\frac{S1 \vdash V_T \qquad S2 \vdash V_E}{\texttt{if c then S1 else S2} \vdash \{(k,v) | v = (mkcond(c)\ ?\ V_T(k)\ :\ V_E(k)) \\ \textit{if } k \in V_T \textit{ and } k \in V_E \\ v = (mkcond(c)\ ?V_T(k)\ :\ k) \\ \textit{if } k \in V_T \\ v = (mkcond(c)\ ?\ k\ :\ V_E(k)) \\ \textit{if } k \in V_E\}}$$

ALLOC: $\texttt{v = new T} \vdash \{(v, DefaultVal)\}$

COLLADD:

$$\frac{f_1.....f_n \text{ are primitive columns in } TypeElement(v)}{\texttt{v.add(w)} \vdash \{(v, v \cup (w\_f_1, ..., w\_f_n))\}}$$

REPOSAVE:

$$\frac{f_1.....f_n \text{ are primitive columns in } Type(v)}{\texttt{repo.save(v)} \vdash \{(repo, repo - \sigma_{id=v\_id}(repo) \cup \\ (v\_f_1, ..., v\_f_n))\}}$$

DELETE: $\texttt{repo.deleteById(v)} \vdash \{(repo, repo - \sigma_{id=v}(repo))\}$

Figure 3: Inference rules for simple statements

we treat the method `setUserDefaultPayment` as if it is the controller in this illustration (although method `setDefaultPayment` is the actual controller). Also, we treat the getter and setter calls in Lines 11, 12, and 15 as if they were inlined to yield direct field references. Each row in Figure 4 depicts the inferred summary for a certain AST subtree, which corresponds to the code region whose line numbers are given in the first column. Each summary is in general a mapping from variables that are modified in the code region to their individual summaries, each of which is a relational algebra expression. Relational algebra expressions use program variables, repository variables or table names, and constants, as leaves.

The row for Line 12 in Figure 4 depicts how assignment statements are modeled: the summary of the LHS variable is simply the expression that occurs in the RHS. The summary for Line 13 uses an extended relational algebra operator, namely, *save*. This operator has two operands, namely, the repository (and underlying table) to save into, and the tuple to save. Note that the single variable *uPIter* in Line 13 in the code has become a tuple of variables at this point; this happens because *uPIter* is an object reference, and has been flattened. Another noteworthy point in Line 13 is that a variable can occur on both sides of the '$\mapsto$' symbol in a summary, with the RHS (resp. LHS) occurrence denoting the incoming value into (resp. outgoing from) the corresponding code region.

The entry for Lines 12-13 in Figure 4 illustrates summary inference for statement sequences. The mappings (i.e., summaries) corresponding to the two regions are basically *composed*; hence, in the example, the second component in the tuple within the *save* operation is now *true* (rather than *uPIter_defaultPayment*).

The entry for Lines 11-17 illustrates processing for if-then-else constructs. A ternary "? :" operator is used in our extended relational algebra to model this construct in a straightforward manner.

Spring ORM implicitly provides schema-specific query methods on repository variables. Our approach converts calls to these methods to equivalent relational algebra. For instance, for the statement in Line 9 in Figure 1, the summary we infer is "$upList \mapsto upRepo$". In cases where a query method returns a single element, summary inference also accounts for flattening. For instance, a statement "`v = uPRepo.findById(x)`" would result in a summary as depicted below:

$$\begin{array}{lcl} v\_id & \mapsto & \Pi_{id}(\sigma_{id=x}\ uPRepo) \\ v\_defaultPayment & \mapsto & \Pi_{defaultPayment}(\sigma_{id=x}\ uPRepo) \\ v\_type & \mapsto & \Pi_{type}(\sigma_{id=x}\ uPRepo) \end{array}$$

Spring ORM supports certain kinds of annotations to introduce fields within entity declarations. These annotated fields do not correspond to columns in the underlying table, but are pointers that explicitly encode relationships with other entities. Our summary inference approach for query method calls accounts for these as well. For instance, consider a field declared as "`@OneToOne UserBilling uBilling`" within the entity class 'UserPayment'. Say `name` is a primitive column in the 'userBilling' table, and say `paymentId` is another column in the 'userBilling' table and is a foreign key into the 'userPayment' table. Say the controller had a statement "`payment = uPRepo.findById(x)`". Our flattening approach treats OneToOne and ManyToOne fields similar to pointers, and flattens through them. Therefore, the statement above would yield a set of statements, one of which would be "*payment_uBilling_name* = `uPRepo.findById(x)`". Our approach generates a summary for this statement in which *payment_uBilling_name* is mapped to the relational algebra expression $\Pi_{name}(\Pi_{Cols(userBilling)}(\sigma_{id=paymentId}(\sigma_{id=x}(uPRepo) \times userBilling)))$. In a similar manner, we handle 'OneToMany' and 'ManyToMany' annotations as well.

A set of formal inference rules to summarize query-method calling statements is given in Figure 5. The bottommost two rules are the *root* rules. In these rules' consequents, the part before the '$\vdash$' is the (flattened) statement that needs to be summarized, with *accp* being a flattened access path. *relExpFor* is a subroutine whose implementation is not shown; it is assumed to return the relational algebra expression corresponding to its argument. The '$\twoheadrightarrow$' derivations are defined using the first three rules in Figure 5. $e_1 \bowtie e_2$ is a

Line #	Summary
12	$uPIter\_defaultPayment \mapsto true$
13	$uPRepo \mapsto save(uPRepo, (uPIter\_id, uPIter\_defaultPayment, uPIter\_type))$
12-13	$uPIter\_defaultPayment \mapsto true$ $uPRepo \mapsto save(uPRepo, (uPIter\_id, true, uPIter\_type))$
15-16	$uPIter\_defaultPayment \mapsto false$ $uPRepo \mapsto save(uPRepo, (uPIter\_id, false, uPIter\_type))$
11-17	$uPIter\_defaultPayment \mapsto (uPIter\_id = defPayId)\ ?\ true : false$ $uPRepo \mapsto (uPIter\_id = defPayId)\ ?$ $save(uPRepo, (uPIter\_id, true, uPIter\_type)) : save(uPRepo, (uPIter\_id, false, uPIter\_type))$
10-18	$uPRepo \mapsto (uPRepo - uPList) \cup \Pi_{id,true,type}(\sigma_{id=defPayId}\ uPList) \cup \Pi_{id,false,type}(\sigma_{id \neq defPayId}\ uPList)$ $uPList \mapsto \Pi_{id,(id=defPayId)?true:false,type}\ uPList$
9-18	$uPRepo \mapsto \Pi_{id,true,type}(\sigma_{id=defPayId}\ uPRepo) \cup \Pi_{id,false,type}(\sigma_{id \neq defPayId}\ uPRepo)$ $uPList \mapsto \Pi_{id,(id=defPayId)?true:false,type}\ uPRepo$

Figure 4: Illustration of summary construction

$$\frac{f_2 \text{ is a *ToOne field} \quad \langle f_2\_f_3\_f_4..., \Pi_{Cols(Type(f_2))}(relExp \bowtie Type(f_2))\rangle \twoheadrightarrow relExp1}{\langle f_1\_f_2\_f_3\_f_4..., relExp\rangle \twoheadrightarrow relExp1}$$

$$\frac{f_2 \text{ is a *ToMany field}}{\langle f_1\_f_2, relExp\rangle \twoheadrightarrow \Pi_{Cols(TypeElement(f_2))}(relExp \bowtie Type(f_2))}$$

$$\frac{f_2 \text{ is a primitive field}}{\langle f_1\_f_2, relExp\rangle \twoheadrightarrow \Pi_{f_2}(relExp)}$$

$$\frac{\langle accp, relExpFor(repo.findByCol(X))\rangle \twoheadrightarrow relExp}{\texttt{accp = repo.findByCol(X)} \vdash \{(accp, relExp)\}}$$

$$\texttt{accp = repo.findAll()} \vdash \{(accp, relExpFor(repo.findAll()))\}$$

Figure 5: Inference rules for query-method calls

join operation whose join predicate equates the foreign-key field in e_2 with the primary key field field in e_1.

2.3 Summary inference for loops

In this part of the paper we discuss how we infer the summary of a loop after having inferred the summary of the body of the loop using pattern-based *rewriting rules*.

In Figure 2, for simplicity of presentation, the LHS of the rewrite rule was shown as if it was a syntactic or code-based pattern. However, in our system, actually, any rewrite rule i is of the form $LHS_i \Rightarrow RHS_i$, where both LHS_i and RHS_i are relational algebra expressions. Both these expressions are patterns, and are hence allowed to have *meta variables*, also known as pattern variables, at leaf (i.e., operand) positions.

We made the important choice mentioned above for a couple of reasons. Firstly, as our summary inference approach proceeds bottom-up in the AST, the relational algebra expression for a loop's body would anyway be available by the time the loop's summary is to be inferred. Secondly, relational algebra summaries are free of internal data flow through local variables, and are hence more declarative. Thus, a pattern based on relational algebra is more likely to successfully match a variety of different loop structures that have the same semantics.

Algorithm 1 SummarizeLoop(*body*, *coll*, *itr*)

1: $summ = \emptyset$
2: **for all** variables v in the domain of *body*, excluding variables of the form itr_f **do**
3: Let $expr_1 = body(v)$
4: **if** exists a rewrite rule $LHS_i \Rightarrow RHS_i$ such that $match(LHS_i, expr_1, coll, itr, v) = v2e$ and $v2e \neq \perp$ **then**
5: Let $expr_2$ be equal to $subst(RHS_i, v2e)$, and $expr_3$ be equal to $expr_2$ with each itr_f_i replaced with $coll.f_i$.
6: Add $v \rightarrow expr_3$ to *summ*
7: **else**
8: Add $v \rightarrow Unknown$ to *summ*
9: **end if**
10: **end for**
11: **if** any itr_f_i is in the domain of *body* **then**
12: Let *tuple* be the tuple formed from the expressions that the variables of the form itr_f_i are mapped to in *body*. Add $coll \rightarrow \Pi_{tuple}(coll)$ to *summ*.
13: **end if**
14: **return** *summ*

The approach for summarizing a loop is depicted in Algorithm 1. The argument *body* is the summary of the loop body alone, obtained previously as part of the bottom-up traversal of the AST. In other words, it is a mapping from each variable that is modified within the loop body to the relational-algebra summary of the value that is assigned to the variable in the loop body (i.e., intuitively, in a single iteration of the loop). The argument *coll* is the name of

the collection variable being iterated over in the loop, while the argument *itr* is the name of the iterator variable. For the loop in Figure 1, *body* is depicted next to Lines 11-17 in Figure 4, *coll* is 'uPList' and *itr* is 'uPIter'. *summ* is the summary of the entire loop, to be returned at the end.

The loop in Lines 2-10 in Algorithm 1 summarizes each variable v one after the other. $expr_1$ is the summary of the value assigned to v in the loop's body. The function *match* is used to check if the LHS of a rewrite rule matches $expr_1$. The function checks the following two conditions: (i) LHS_i and $expr_1$ are isomorphic as trees, not counting the operand positions in LHS_i where there are meta variables, and (ii) at every position where the *special* meta variables $\langle coll \rangle / \langle itr \rangle / \langle lvar \rangle$ occur in LHS_i, the values of the arguments $coll/itr/v$, respectively, occur in $expr_1$. If the check passes, *match* returns a mapping *v2e*, which maps every meta-variable occurring in LHS_i to its matching sub-expression in $expr_1$. *v2e* also maps $\langle coll \rangle / \langle itr \rangle / \langle lvar \rangle$ to the values in arguments $coll/itr/v$, respectively. *match* returns $\perp$ if the conditions named above do not hold. If the LHS's of multiple rules match $expr_1$, Line 4 in the algorithm picks the earliest matching rule.

Lines 5-6 in the algorithm produce the summary of v as far the full loop is concerned; basically, this summary accounts for the cumulative updates done to v across all iterations of the loop. The summary is produced by the function *subst*, which simply replaces every occurrence of any meta variable w in RHS_i with the corresponding subexpression $v2e(w)$. Each occurrence of itr_f_i is finally replaced with $coll.f_i$, as itr_f_i is a local variable and represents the value of the field f_i in a single iteration.

2.3.1 Illustrations. Part (A) of Figure 6 illustrates the summarization of the loop in Figure 1. Row (b) depicts the LHS of a rewriting rule, while Row (c) depicts the RHS. (These correspond to the LHS-RHS shown informally as a code-pattern in Figure 2.) The variable whose summary is being computed (v in Algorithm 1) is 'uPRepo'. Row (a) depicts *body*(uPRepo), which is the summary of 'uPRepo' as inferred from the loop's body. The mapping *v2e* is represented by the underbraces. Row (d) depicts the result of applying *subst* on the RHS pattern in Row (c) using the mapping *v2e* referred to above. This result is the final relational expression for 'uPRepo' in the loop's summary (the same information is present against Lines 10-18 in Figure 4). Intuitively, this rewrite rule is meant to summarize the action of saving different tuples into a repository across different iterations of a loop.

Lines 12-13 in Algorithm 1 basically produce the summary for 'uPList' that is shown against Lines 10-18 in Figure 4).

Rows (b) and (c) in Part (B) of Figure 6 depict another sample rewrite rule. This rule is meant to summarize the action of deleting specific tuples from a repository in each iteration of the loop. Rows (a) and (d) show the pre-computed loop-body-summary and the full-loop-summary inferred using the rewrite rule, respectively, for the example loop that appears to the left in Part (B).

Finally, Part (C) follows a similar format as Part (B), and illustrates a another sample rule, which infers a sum-aggregation operation over a field of all entities in a collection. This rule can be generalized easily to account for the situation where only certain entities in the collection are selected for the aggregation.

2.4 Capabilities and limitations

Our approach handles nested loops naturally. We handle precisely only loops that iterate over collections, as ORM programs primarily use this type of loop. For other types of loops we conservatively map modified variables to '*Unknown*' values. Our approach performs inter-procedural analysis basically by simulating inlining. This technique may not terminate in the presence of recursion, but we have not come across recursion in application code in Spring benchmarks that we have seen. The two limitations just mentioned are also shared by closely related work [2].

In our approach, the summary of a controller not only includes summaries for updated repositories, but also summaries for *model attributes*, which are the return values sent by controllers to views.

Our approach is *sound* with regard to loop-free code fragments. Soundness means that any property that is implied by a summary is also satisfied by the code. Our approach is also *complete* with regard to loop-free code fragments, in the sense that if relational algebra suffices to capture the full semantics of a fragment of code, our approach will infer such a summary. Both these claims are conditional on no-aliasing between variables and on a sufficient bound for flattening.

A limitation of our approach is that if a variable or access path refers to a collection of entities, then the summary of this variable or access path will be a 'σ' expression that contains information about primitive fields but contains no information about Spring-annotation fields within these entities. Capturing such information in general needs *nested* relational algebra. Currently our inferred summaries as well user-provided properties are restricted to *flat* relational algebra (in which only primitive fields in collections are referred to).

A final point to note is that in the presence of loops our approach does not have absolute soundness or completeness. Currently it is up to developers to ensure that the rewrite rules they specify are sound, i.e., semantically valid. Also, the extent to which loops are summarized precisely depends on the sufficiency of the set of rewrite rules provided.

3 CHECKING PROPERTIES OF TRACES

Generally, many interesting properties of controllers are in terms of their incoming and outgoing database states (in addition to the input arguments and returned model attributes). For instance, in an open-source benchmark called 'PetClinic' [14] that we used in our evaluations, there is a controller named 'processCreationForm'. It accepts a tuple of (primitive typed) input arguments corresponding to an 'Owner' entity, such as (id, lastName, firstName, city), and saves this tuple as an entity into a repository called 'owRepo'. Say the property that the developer has in mind is to check if this controller indeed saves the given tuple into persistent state. To write this as a single-controller property, the developer would first need to guess that this controller would be saving its argument tuple into some database table, and would secondly need to know the name of repository it is saving into. We also refer the reader to the sample property in Equation (2) in Section 1.2, which has a similar flavor, and is in terms of the updated state of the repository 'uPRepo'.

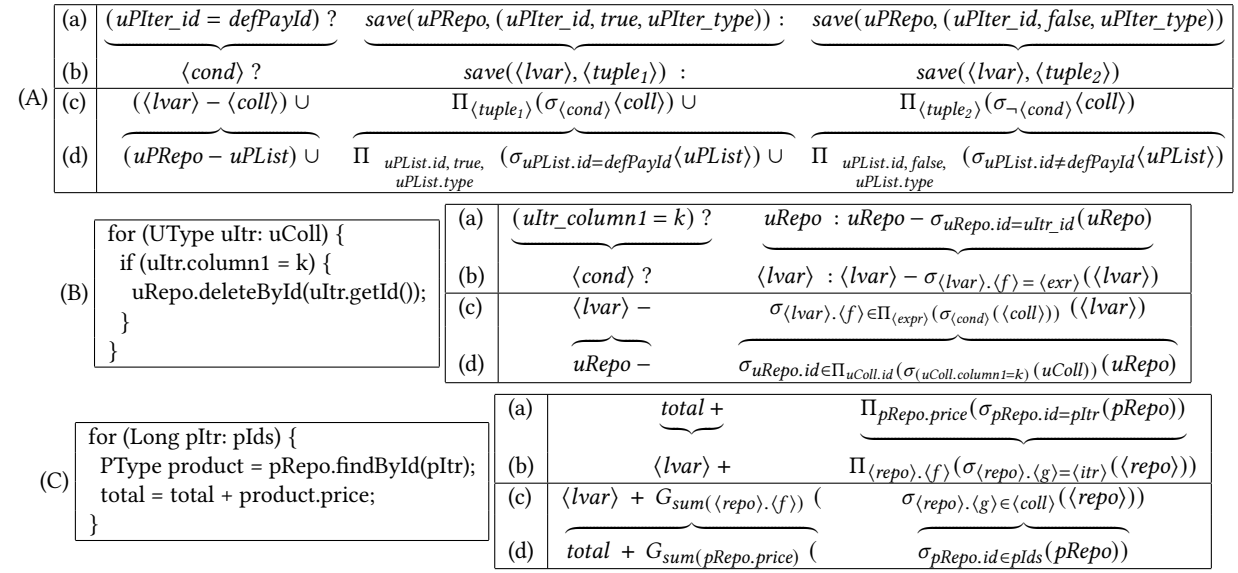

Figure 6: Sample loop-summarization patterns

Our observation in this section is that awareness of the database schema and of the access relationships between controllers and database tables may become unnecessary if properties are specified in a different way – purely in terms of argument and return-value behavior of *pairs* of related controllers. For instance, in the PetClinic benchmark there is another controller named 'processFindForm', which takes a lastName as argument, and returns (to a view) via a model attribute the set of all Owner entities with the given lastName. If one is aware of this controller and its I/O behavior as stated above, one could write more natural (i.e., database independent) version of the single-controller property mentioned in the previous paragraph as follows:

If processCreationForm is invoked in a trace and then processFindForm is invoked next, if the lastName given to processFindForm is equal to the lastName given to processCreationForm, then one of the 'Owner' entities returned by processFindForm agrees in all its fields with the tuple of argument values given to processCreationForm.

3.1 Approach

Definition 3.1 (Trace property). A trace property wrt two controllers C_A and C_B is a predicate $\psi(i_f, o_f, i_l, o_l)$ on the variables i_f, o_f, i_l, o_l, where i_f represents the tuple of input arguments to C_A, o_f is the tuple of return values (i.e., model attributes) from C_A, and i_l and o_l are analogously defined and pertain to C_B.

Definition 3.2 (Trace satisfaction). A trace t (i.e., a run of the application) is said to satisfy a trace property $\psi(i_f, o_f, i_l, o_l)$ wrt two given controllers C_A and C_B and wrt a given set of controllers *notBetween* if the following condition holds: *If t invokes C_A at some point with actual arguments i_a and receives actual return values o_a, and t invokes C_B at some later point with actual arguments i_b and receives actual values o_b, and t does not visit any controller in the set* notBetween *between the aforementioned visits to C_A and C_B, then $\psi(i_a, o_a, i_b, o_b)$ holds.*

Our **problem statement** is: Given two controllers C_A and C_B and a set of controllers *notBetween* and a trace-property ψ, check if *all traces* of the application satisfy ψ.

Note that we have generalized our problem statement, in that C_A and C_B need not be invoked back to back. This generalization gives a stronger guarantee about the application, as it reasons across a potentially infinite set of traces of unbounded lengths.

We use the notation $C(d_i, d_o, i_a, o_a)$ to denote the summary of a given controller C as inferred by the approach of Section 2. d_i and d_o are variables in the summary that denote the incoming data base state and outgoing data base state, respectively, while i_a and o_a denote the input arguments to and return values from the controller, respectively.

The approach we propose for our problem is basically to check the following two properties using any relational algebra solver.

$$\begin{aligned} &\forall d_1, d_2, d_3, i_A, i_B, o_A, o_B \\ &C_A(d_1, d_2, i_A, o_A) \wedge C_B(d_2, d_3, i_B, o_B) \\ &\quad \Rightarrow \psi(i_A, o_A, i_B, o_B) \end{aligned} \tag{A}$$

Intuitively, Property (A) above checks that ψ is satisfied whenever controller C_B is invoked directly after C_A in any trace.

$$\begin{aligned} &\forall C_X \notin notBetween \\ &\forall d_1, d_2, d_3, i_A, o_A, i_B, o_B, d_X, i_X, o_X, i_4, o_4 \\ &\left[\begin{array}{lcl} \Big(C_B(d_1, d_2, i_B, o_B) & \Rightarrow & \psi(i_A, o_A, i_B, o_B)\Big) \\ & \Longrightarrow & \\ \left\{ \begin{array}{c} C_X(d_1, d_X, i_X, o_X) \\ \wedge \\ C_B(d_X, d_3, i_4, o_4) \end{array} \right\} & \Rightarrow & \psi(i_A, o_A, i_4, o_4) \end{array} \right] \end{aligned} \tag{I}$$

Property (I) is actually a template for a *set* of properties, one for every $C_X \notin notBetween$. The property above basically checks that if

a trace ends at C_B and the trace satisfies ψ, then upon inserting an invocation to C_X just before C_B the resultant trace also satisfies ψ. That is, an invocation to C_X does not interfere with the satisfaction of the property.

Intuitively, the approach solves our problem in a sound manner for the following reason. Property (A) above is the base case, and discharges correctness for traces that don't visit any controller between C_A and C_B. Property (I) is the inductive case, and basically implies that any sequence of visits to controllers that are not in *notBetween* can be inserted between C_A and C_B without interfering with the property. A detailed proof of soundness is included in a supplementary document *other-details.pdf* [7] associated with this paper.

3.1.1 Illustration. If C_R denotes the controller processCreationForm and C_F denotes processFindForm, then the inferred summaries would be as follows:

C_R(*owRepo, owRepo'*, (*id, lastName, firstName, city*), _) ≡
  owRepo' = *save*(*owRepo*, (*id, lastName, firstName, city*))
C_F(*owRepo, owRepo, lastNameX, ret*)) ≡
  $ret = \sigma_{lastNameX=owRepo.lastName}(owRepo)$

Note, we use *owRepo'* to refer to the outgoing state of this repository in order to treat the summary as a formula rather than as a mapping.

The trace property given by the developer could be:

ψ((*id, lastName, firstName, city*), _, *lastNameX, ret*) ≡
  (*lastName* = *lastNameX*) ⟹
  (*id, firstName, lastName, city*) ∈ *ret*

The developer may indicate *notBetween* to contain all controllers that they believe do not delete or update any 'Owner' entity in the persistent state.

3.2 Capabilities and limitations

To our knowledge, our proposal above is the first one to use an efficient, inductive approach to check properties of *all* traces in a web application without bounding the lengths of traces. The number of properties checked by the approach using calls to the solver is linear in the number of controllers in the application.

Our approach is sound if the individual controller summaries are sound; i.e., the approach will not declare a property that does not hold as holding. The approach can suffer from false positives. The fundamental cause is that the approach ignores the effects due to views, which can restrict the order in which controllers may be invoked, and can also restrict what data flows in as arguments to a controller. Currently, the given property ψ can refer to two controllers C_A and C_B. An extension to more than two controllers (but a fixed number of them) is conceptually straightforward, and provided in our supplementary document *other-details.pdf* [7].

4 IMPLEMENTATION

We have implemented our controller summarization approach (described in Section 2) as a prototype tool called *ORMInfer*. The tool is implemented using the Soot bytecode analysis framework [26]. Our summary-construction code is based on the DBridge [5] code, with many additional features added (which are summarized in Section 6). The default flattening length bound in our tool (see Section 2.1) is three (i.e., three underscores). We have identified a subset of commonly used library functions, and translated them directly into relational algebra during summary inference. Any other library calls, if encountered, are treated, in the interest of efficiency, as if they return arbitrary values.

One of the major new components in our implementation over DBridge is the one that accepts pattern-based rewrite rules, and applies them during analysis time. Our tool provides a simple custom DSL (domain specific language) for specifying these rewrite rules. For instance, for a simplified version of the rewrite rule in Figure 6(A), the corresponding rule in the DSL would be as follows:

```
(loop (bodyexpr (? <cond> (save <lvar> <tuple1>)
  (save <lvar> <tuple2>)))  <lvar> <coll> <itr>)
(union (- <lvar>  <coll>)
               (union (pi (select <coll> <cond>) <tuple1>)
                  (pi (select <coll> (= <cond> 0)) <tuple2>)))
```

The DSL uses a Lisp-like prefix notation, with keywords/operators preceding operands. The first two lines above encode the LHS; actually, the sole operand of `bodyexpr` keyword represents the LHS pattern. The three names towards the end of the `loop` construct are the names of the special meta variables introduced in Section 2.3 (these meta variable names are not fixed, and can be chosen by the pattern specifier). The last three lines above represent the RHS of the pattern. We include five specific rewrite rules with the tool, and more can be added by users. We provide these five patterns in the supplementary document *other-details.pdf* [7].

We currently use Alloy [18] as our backend tool for checking properties. To this end, we have implemented a postpass that takes a relational algebra summary of a controller (in memory) and translates it into an Alloy model. Since Alloy itself is based on relational algebra, the translation is defined quite naturally. Every variable in the domain of a controller's summary (i.e., updated repositories, assigned model attributes) becomes a "sig" in the Alloy model, and the model contains facts that are translations of the relational algebra expressions that the variables are mapped to. Assertions can subsequently be added to the Alloy model by users. The assertions can refer to the "sig"s mentioned above, and can be checked by the Alloy tool. Currently, we abstract away any scalar arithmetic that may be present in the summary and replace these subexpressions with unconstrained values.

We have also implemented the trace-property checker described in Section 3 as a tool *MultiORM*. This tool uses the relational algebra summaries inferred by ORMInfer for the individual controllers to emit the Property (A) and a set of Property (I)'s in Alloy form for each given trace property.

5 EMPIRICAL EVALUATION

This section describes the initial empirical evaluations we have performed using our prototype tools to evaluate their usefulness, precision, and efficiency.

5.1 Benchmarks and Properties

We selected six open-source benchmarks for our evaluations. Key statistics about the benchmarks are summarized in Table 1. Our key criteria for choosing a benchmark were that it should use Spring ORM features for data access, and should not use third-party libraries or frameworks extensively. Many of the benchmarks in

Benchmark	Controllers	Entities	Java LOC	Github Stars	URL
PetClinic	17	6	2762	4955	[14]
Spring Boot Blog	14	4	1204	131	[12]
Employee Directory	5	1	375	0	[10]
Imagine	22	4	3538	17	[11]
Spring Coffee Shop	6	3	372	7	[13]
Bookstore	24	12	2892	0	[9]

Table 1: Benchmark statistics

Category	Single	Trace
True Neg.	44	28
True Pos.	2	7
False Pos.	1	7
Unexpressed	12	4
TOTAL:	59	46

Table 2: Usefulness and precision results

our list were created by the Spring community to illustrate the ideal usage of Spring features to build realistic applications.

Our next step was to obtain a set of properties for evaluation. Since the benchmarks come with almost no assertions in the code, we approached volunteers we knew and asked them to understand the benchmarks and provide us properties (or specifications) for us to check. These volunteers were either PhD students or post-docs, were experienced in programming, and had good familiarity with notions such as property checking, first-order logic, relational algebra, etc. However, they were unfamiliar with our work and with the abilities of our approach. There were a total of four volunteers. We requested each volunteer to supply properties for three benchmarks, so each benchmark had properties from two volunteers. We asked each volunteer to give us *single-controller* properties, e.g., similar to Property (1) in Section 1.1, as well as *trace properties*, e.g., similar to the 'ψ' in Section 3.1.1, for all their benchmarks.

In order to make the work of the volunteers easier, we gave them various resources, such as: (i) database schemas, to enable them to write single-controller properties, (ii) hosted instances of the applications, to enable them to use the applications and come to understand them well, and (iii) a few sample properties (of both types). The volunteers did not look at the source codes of the benchmarks. An important guideline given to them was to write properties based on *expected* behavior from the end-user perspective, even if any bugs in the applications resulted in non-expected behavior.

We asked the volunteers to give us each property in English wording. We decided that asking them to formally state the properties might prove too burdensome and might disincentivize them. In the remainder of this section we present the results from our evaluations. We have made available a virtual machine image [7] that contains our tools, scripts to run them, as well as inputs necessary to reproduce all results given in this section.

5.2 Performance on single-controller properties

We first went through the given single-controller properties to translate them manually to Alloy assertions using our best judgment. We had to "reject" 14 of the given properties – four because they were too vague, five because they were trivially implied by the constraints of the schema, and the remaining five for one-off reasons that would need more space to explain. We also ignored "repeat" properties – i.e., essentially duplicates of properties provided by other volunteers. What remained were 59 properties. We inferred summaries in Alloy form for the controllers to which these properties pertained using ORMInfer, and then used the Alloy tool to check if the summaries implied the properties. The results are summarized in the "Single" column of Table 2.

A *negative* means the property was found to pass by the tool (i.e., the Alloy assertion did not fail), while a *positive* is the converse. *True* means the tool's decision agrees with our understanding of the application's behavior, while a *False* means the converse. "Unexpressed" means that although the summary (and its corresponding Alloy model) were generated by the tool, the summary did not contain certain elements that are necessary to translate the property into an Alloy assertion. Note, we did not notice any cases of unsoundness, and hence there is no row titled "False Negative".

Overall, the performance of our approach is very good. 46/59 properties (78%) are in the *True* categories. Note that 44/59 properties hold. Two properties (which were on the same controller) actually did not hold: the volunteer expected this controller, which saved a given 'Employee' entity into a database, to check for uniqueness of the given email ID and non-emptyness of other fields. However, it was not doing these checks.

There was only one false positive. It was due to constraints on incoming arguments imposed by preceding views, which were not encoded in the summaries. 7/12 of the "unexpressed" assertions were not expressible because the summary had no information on collection-typed fields which were themselves inside collections; as discussed in Section 2.4, one needs nested relational algebra to represent such summaries. The remaining 5 unexpressed properties were due to other one-off reasons that cannot be explained due to space constraints.

Our approach turns out to be very efficient. On four of the six benchmarks, the maximum analysis time per controller, including summary inference time and assertion checking time using Alloy, was 3 seconds. With 'Bookstore' the maximum was 10 seconds and average was 3 seconds, while with 'Imagine' the maximum time was 104 seconds and average was 17 seconds. In our all runs of the Alloy tool we used a *universe* of 20 elements.

5.3 Performance on trace properties

We followed a similar process as above, beginning with a manual translation of the given English-language properties into Alloy assertions. In this case, we had to "reject" 17 of the given properties. The prevalent reason (accounting for 12/17 properties) was that the property was referring to three different controllers that the trace had to go through. The current limit of our tool is two controllers (see Section 3.2), although in principle it is not difficult to extend our tool to process 3-controller properties. After ignoring "repeat" properties as well, there remained 46 properties that we

handled using our tool. We used our MultiORM tool to generate Alloy facts and assertions corresponding to the Property (A) and the set of Property (I)'s for each given trace property, and checked all these generated assertions using the Alloy tool. The results are summarized in the "Trace" column of Table 2.

Overall, the performance of our approach is very good. 35/46 properties (76%) are in the *True* categories. Seven of the properties failed as per the tool, and truly did not hold in our opinion. Our manual analysis revealed that all these were actually due to the result of oversight or misunderstanding by the volunteers. A prevalent reason was volunteers not putting all the controllers that can indeed be expected to affect the return value from C_B in the set *notBetween*. Note that if a human subject expects a property to hold and it actually does not hold and the tool reports it as not holding, then it really is a *true positive* irrespective of the fact that the subject's expectation was due to a limitation in their understanding.

There are relatively larger number of false positives (7/46) in this part of our evaluation. The reasons are varied, and hard to present in detail. Imprecise handling of certain complex Spring idioms and library calls account for many of these cases. Two of the "Unexpressed" cases were because a field of a result entity was not present in the summary due to its dependence on arithmetic, while two were due to the need for nested relational algebra.

The average time spent by Alloy to check a trace property was 5 seconds, while the maximum was 54 seconds.

5.4 Ability to find bugs

In this part of our study we wish to answer a natural question, namely, whether a reasonably large set of assertions if written apriori would be useful in detecting bugs, including bugs that may get introduced in the future. Since our benchmarks did not possess many bugs at all, we decided to seed *mutations* in our benchmarks. This is a common practice by researchers who wish to evaluate bug-detection approaches. For this, we used the automated production-quality tool PIT (https://pitest.org).

Since this study involved some time-consuming effort, we focused our attention on two benchmarks, namely, PetClinic and Bookstore. From our single-controller study (Section 5.2), we identified all 16 passing (i.e., *True Negative*) assertions, and applied PIT on the controllers tested by these assertions as well as their callees. PIT suggested a total of 33 separate mutations. We applied these mutations one by one in the benchmark codes, and ran our tool separately on each mutated version of the benchmark. 6 of the mutations could not be handled by our tool, because there is currrently a limitation in the tool that prevents it from analyzing callee methods that contain an explicit "return null" statement (these statements got introduced due to the mutations). Of the 27 mutated versions that were analyzed by our tool, 15 caused at least one of the provided assertions to fail (i.e., 15 were "killed").

Our takeaway is that a set of general-purpose assertions written apriori without any regard to any specific bugs still has the potential to find a significant proportion of bugs that could be introduced in the future (based on the 56% kill rate in the evaluation above).

5.5 Comparison with a baseline

Finally, we wished to comparatively evaluate our approach with a baseline tool. There are no comparable controller summarization tools for Java that we are aware of. The closest matching family of tools is *symbolic execution* tools, as they can automatically check assertions. We decided to use the widely-used tool *Java Pathfinder* (JPF), https://github.com/javapathfinder. JPF is not directly meant to test Spring controllers, so we made some manual changes to the benchmarks to make them amenable to analysis by JPF. The main changes were to initialize the database tables with tuples that contained *symbolic values* in order to enhance the coverage of JPF, to call each controller like a normal method, and to write the assertions in Java.

Since this study also needed significant manual effort, we decided to focus only on the 16 assertions mentioned in Section 5.4 above. JPF also found all 16 of these assertions to pass. It took 7 seconds for one of the assertions, and around 1 second each for the rest. As JPF explores execution traces in-depth, we expect it to also declare as passing most of our false-positive assertions.

In other words, JPF is efficient and effective at checking assertions, provided one puts in the manual work as discussed above. However, the utility of our approach is not just in checking assertions, but in producing general-purpose summaries of controllers that are amenable to various different downstream analyses. For instance, the trace-checking application was easily enabled using our summaries. With JPF, only a finite number of traces, of bounded lengths, can be checked.

5.6 Manual intervention during experimentation

It is to be noted that in the studies reported above involving our tool, we manually added some *facts* (i.e., constraints) to the Alloy models of some of the controllers to improve precision. The constraints were on incoming arguments to the controllers, and were meant to either encode knowledge about these arguments derived from preceding views or controllers, or prune out paths within the controller that throw exceptions on ill-formed arguments, etc. Eighteen of the 44 true negative properties in the single-controller experiments and nine of the 28 true negative properties in the trace property experiments needed such manually added constraints. We believe that in real usage, developers would be willing to add such constraints in order to get maximum benefit from the tool.

A limitation of our currently implemented Alloy generation postpass is that "sig"s and fields are emitted only for those parts of the database schema that are actually referred to in the controller. However, assertions sometimes need to refer to schema elements that are not referred to in the controller. This limitation can be removed from the tool in the future, but for now we manually add such required information from the schema into the Alloy models on demand.

Finally, for the trace-checking study (Section 5.3) alone, we modified the benchmarks to simplify two specific idioms that can obstruct the precision of our analysis: Replaced substring matching using "LIKE" with equality in embedded SQL wherever it is present, and replaced calls to *Spring* that return the currently logged in username with a constant username.

6 RELATED WORK

The closest related work to our work is that by Bocić and Bultan [2]. Their approach actually infers a verification condition for checking a property for a controller, but it can be seen as inferring a summary as well. Their summary is represented in FOL (First Order Logic). They use FOL without scalars or arithmetic, so arithmetic operations are abstracted away, as are *all* conditionals. This results in imprecision. Our core summary inference approach (described in Section 2) is based on relational algebra, and precisely represents scalars, arithmetic, conditionals, as well as aggregations over loops. Our current translation of the summaries to Alloy does result in abstracting away of arithmetic, but non-arithmetic based conditionals (such as the one in the running example in Figure 1) are retained, and such conditionals did play a major role in enhancing precision as per our evaluations.

The technique employed by their approach to summarize loops is fully automated, and does not use patterns. However, their approach is efficient and terminates in practice only when there are no loop-carried data flows [3]. With patterns we do not have this restriction, and we are able to summarize precisely loops with loop-carried flows, e.g., ones that add up values from a collection. Also, our approach emits its summaries in Alloy, which is both human readable and amenable to a variety of downstream analysis-based applications.

It was not possible to directly apply their tool on our benchmarks, as their work targets Ruby on Rails applications. They have reported in their paper a significantly lower rate of false positives and "unexpressed" properties than we do. However, it appears that their properties were written by the authors themselves, whereas our properties were provided by volunteers who were not involved in our work. We studied the ten loops that occur across our six benchmarks, and found that our approach infers precise summaries for six of them. Whereas, upon a conceptual and manual application of their approach on our ten loops, we found that their approach would be able to infer a precise summary for only one of the ten loops.

There exists a rich body of work on inferring SQL from imperative code fragments using program analysis, primarily focusing on loops that read from databases [4, 5, 15, 17, 28]. The objective of these works is generally to optimize loops by replacing them with SQL, which can be optimized by query optimizers. The closest work from this body to our work is DBridge [5]. Their approach performs bottom-up summarization of ASTs, and this aspect of our implementation is in fact borrowed from their implementation. They address a fixed set of looping idioms in code. We have generalized this aspect of their work into a generic rewrite system for loops based on developer specified patterns. The use of flattening to address heap references is new in our approach, as is the handling of various Spring-specific features. On the ten loops that we had referred to earlier in this section, DBridge is able to infer precise SQL for just two of them.

A key difference between the approaches mentioned above and ours is our focus on representing *all* the effects of a controller, including database updates and model attributes returned, into the summary, and then using the summary for property-checking purposes. A couple of recent approaches [19, 23] perform black-box analysis of a querying code or query to infer equivalent SQL, with the objective of program understanding, reconstruction, or migration.

Logical methods have been used by researchers to reason about database accessing applications. The work of Itzhaky et al. [16] focuses on computing weakest preconditions in a simple loop-free scripting language that allows embedded SQL. The work of Wang et al. [27] is about proving equivalence of two database-accessing programs written in an intermediate language.

A number of papers propose techniques for automated test-case generation or symbolic execution to find bugs in database-accessing applications [6, 20, 22, 25]. These approaches do not produce summaries of code, but rather explore paths in the code in an attempt to find bugs. The work of Athaiya et al. [1] is orthogonal to ours, in that it focuses on inferring summaries of views rather than controllers.

7 CONCLUSIONS AND FUTURE WORK

We presented in this paper a novel, pattern-based approach for summarization of ORM controllers. We explored in-depth applications of the inferred summaries to property checking for controllers. Our implementation of our approach was very efficient, and showed promising precision, with around 78% of the properties processed with correct results. We not only checked properties of individual controllers using our summaries, but also showed an application or extension of these summaries to check all possible traces in the application to see if they satisfy a specific kind of trace property.

Our work opens up several ideas for future work. We could expand the set of controllers that get summarized precisely by incorporating nested relational algebra, and by incorporating features such as "group by" and "having". Constraints on incoming arguments to controllers could be inferred from preceding views and preceding controllers. Checking soundness (i.e., semantics preservation) of a given rewrite rule could be a very interesting problem. We could potentially make use of points-to analysis in order to eliminate the potential for unsoundness in the current flattening approach in the presence of aliasing. Finally, we suggest that other applications of the inferred summaries be explored, such as automated comparison or merging of different versions of the same controller's code, automated test input generation, etc.

ACKNOWLEDGMENTS

This work was made possible by generous research sponsorship from TCS Limited, by scholarships from the Ministry of Education, Govt. of India, and by financial support from IBM Research India. We thank Alvin George, Habeeb P, Rekha Pai, and Stanly Samuel, for providing the properties for our evaluations.

REFERENCES

[1] Snigdha Athaiya and Raghavan Komondoor. 2017. Testing and analysis of web applications using page models. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 181–191.

[2] Ivan Bocić and Tevfik Bultan. 2014. Inductive verification of data model invariants for web applications. In *Proceedings of the 36th International Conference on Software Engineering*. 620–631.

[3] Ivan Bocić and Tevfik Bultan. 2015. Coexecutability for efficient verification of data model updates. In *2015 IEEE/ACM 37th IEEE International Conference on*

Software Engineering, Vol. 1. IEEE, 744–754.

[4] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices* 48, 6 (2013), 3–14.

[5] K Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S Sudarshan. 2016. Extracting equivalent SQL from imperative code in database applications. In *Proceedings of the 2016 International Conference on Management of Data*. 1781–1796.

[6] Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. 151–162.

[7] Chawla et al. 2022. Supplementary materials. https://doi.org/10.6084/m9.figshare.19087814

[8] Roy T Fielding and Richard N Taylor. 2002. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)* 2, 2 (2002), 115–150.

[9] Github. 2022. Bookstore. https://github.com/justBrokkoly/bookstore.git

[10] Github. 2022. Employee. https://github.com/kiticgoran90/crud-employee-thymeleaf/

[11] Github. 2022. Imagine. https://github.com/yyqian/imagine.git

[12] Github. 2022. Spring Boot Blog. https://github.com/reljicd/spring-boot-blog.git

[13] Github. 2022. Spring Coffee Shop. https://github.com/shakeelosmani/springcoffeeshop

[14] Github. 2022. Spring Petclinic. https://github.com/spring-projects/spring-petclinic

[15] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 559–573.

[16] Shachar Itzhaky, Tomer Kotek, Noam Rinetzky, Mooly Sagiv, Orr Tamir, Helmut Veith, and Florian Zuleger. 2017. On the Automated Verification of Web Applications with Embedded SQL. In *20th International Conference on Database Theory, ICDT*. 16:1–16:18.

[17] Ming-Yee Iu, Emmanuel Cecchet, and Willy Zwaenepoel. 2010. JReq: Database queries in imperative languages. In *International Conference on Compiler Construction*. Springer, 84–103.

[18] Daniel Jackson. 2019. Alloy: A Language and Tool for Exploring Software Designs. *Commun. ACM* 62, 9 (Aug. 2019), 66–76. https://doi.org/10.1145/3338843

[19] Kapil Khurana and Jayant R Haritsa. 2021. Shedding Light on Opaque Application Queries. In *Proceedings of the 2021 International Conference on Management of Data*. 912–924.

[20] Joseph P Near and Daniel Jackson. 2012. Rubicon: bounded verification of web applications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[21] Stack Overflow. 2020. Stack Overflow Developer Survey 2020. https://insights.stackoverflow.com/survey/2020#technology-web-frameworks

[22] Filippo Ricca and Paolo Tonella. 2001. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE, 25–34.

[23] Jiasi Shen and Martin C Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 269–285.

[24] Pivotal Software. 2022. Spring Framework. https://spring.io/projects/spring-framework

[25] Suresh Thummalapenta, K Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. 2013. Guided test generation for web applications. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 162–171.

[26] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[27] Yuepeng Wang, Isil Dillig, Shuvendu K Lahiri, and William R Cook. 2017. Verifying equivalence of database-driven applications. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

[28] Ben Wiedermann, Ali Ibrahim, and William R Cook. 2008. Interprocedural query extraction for transparent persistence. *ACM Sigplan Notices* 43, 10 (2008), 19–36.