Systematic Generation of Non-Equivalent Expressions for Relational Algebra

Kaiyuan Wang[1], Allison Sullivan[1], Manos Koukoutos[2], Darko Marinov[3], and Sarfraz Khurshid[1]

[1]University of Texas at Austin, USA
[2]École Polytechnique Fédérale de Lausanne, Switzerland
[3]University of Illinois at Urbana-Champaign, USA
kaiyuanw@utexas.edu, allisonksullivan@utexas.edu, emmanouil.koukoutos@epfl.ch, marinov@illinois.edu, khurshid@utexas.edu

Abstract. Relational algebra forms the semantic foundation in multiple domains, e.g., Alloy models, OCL constraints, UML metamodels, and SQL queries. Synthesis and repair techniques in such domains require an efficient procedure to generate (non-equivalent) expressions subject to relational constraints, e.g., the types of sets and relations, their cardinality, size of expressions, maximum arity of the intermediate expressions, etc. This paper introduces the first generator for relational expressions that are non-equivalent with respect to the semantics of relational algebra. We present the algorithms that define our generator, its embodiment based on the Alloy tool-set, and an experimental evaluation to show the effectiveness of its non-equivalent generation for a variety of problems with relational constraints.

1 Introduction

Relational algebra forms the semantic foundation in multiple domains, e.g., Alloy models [16], OCL constraints [39], UML metamodels [42], and SQL queries [25]. Developing program synthesis [6, 14, 22, 36, 37, 46] or program repair [10, 20, 23, 24, 40, 57] methods in such domains requires an efficient technique to generate (non-equivalent) expressions subject to relational constraints, e.g., the types of sets and relations, their cardinality, size of expressions, maximum arity of the intermediate expressions, etc.

While syntactically different expressions can be generated by simple standard bottom-up [54] or top-down [6] grammar-based generation techniques, generating expressions that way can produce an infeasibly large number of expressions even for relatively small expression sizes. (In this paper, we measure the size of an expression by the number of AST nodes in that expression.) For example, for just one binary relation $r \subseteq S \times S$ on some set S and expression size up to 7, there are 17109 syntactically different expressions that can be built using the operators from Alloy [16]: 5 standard binary operators (relational join, Cartesian product, set union, intersection, difference) and 3 unary operators (transpose, transitive closure, reflexive transitive closure). Reducing this number of expressions requires reasoning about semantic equivalences in relational algebra. Some such equivalences are well-known, e.g., associativity and commutativity (AC) for set union and intersection. While AC rules reduce the number of expressions to 11191 in this example, there are actually only 771 non-equivalent expressions. We need more advanced equivalences to prune out equivalent expressions.

We introduce RexGen, the first generator for semantically non-equivalent relational expressions. We present the algorithms that define our generator, its embodiment based on the Alloy tool-set, and an experimental evaluation to show the effectiveness of its non-equivalent generation for a variety of problems with relational constraints.

Our choice of Alloy is driven by its foundation in relational, first-order logic [16], its focus on analyzability, and its wide application in various domains, e.g., software design [7,17], analysis [4,9,18], testing [31], and security [21]. Alloy's tool-set includes an automatic analysis tool [53] for checking the satisfiability of formulas written in Alloy using off-the-shelf propositional satisfiability (SAT) solvers. The analyzer performs *scope-bounded* analysis, which checks the properties within a given *scope*, i.e., bound on the universe of discourse. While the Alloy analyzer could be used to check semantic equivalences of all expressions during generation, it results in an impractically slow generation.

Our key insight is that the Alloy analyzer enables a systematic method for creating and evolving an *optimized* generator for non-equivalent relational expressions. Our method first uses the analyzer (with its expensive, semantic equivalence checks) to discover likely equivalences of expressions that already get generated. We then generalize and validate these likely equivalences using manual reasoning, and incorporate them in the expression generator as *equivalence rules*. These rules directly prune equivalent expressions based on quick, (mostly) *syntactic* checks without expensive, semantic equivalence checks.

RexGen offers three automatic pruning modes for bottom-up generation of relational expressions. One mode, *static* pruning, directly prunes from generation many equivalent expressions based on a fixed suite of equivalence rules, which include well-known equivalences and also dozens more that we discovered using the Alloy analyzer. Another mode, *dynamic* pruning, uses the analyzer during generation to prune equivalent expressions incrementally by comparing each new expression to a representative from each equivalence class formed thus far, while forming new equivalence classes as needed. The third mode, *modulo-instance* pruning, allows the user to provide AUnit *test* valuations [50, 51], and prunes an expression if it is equivalent to some generated expression with respect to all given test valuations (even if not equivalent for some other valuations [3]).

We perform an experimental evaluation of RexGen using expression generation problems derived from 12 Alloy models. We evaluate the number of expressions that RexGen generates and the time that RexGen takes to generate those expressions for each problem under different settings. The experimental results show that static pruning offers the best trade-off, creating mostly *semantically different* expressions, substantially reducing the number of expressions from simple grammar-based generation, while not increasing the generation time—in fact, often having smaller generation time than not using any equivalence pruning rules. In comparison, using only AC rules, as done by some state-of-the-art systems for expression generation [23] (albeit not for relational expressions, so we added appropriate extensions for comparison), generates a larger number of expressions, while not substantially reducing the time. Using dynamic pruning removes all equivalent expressions w.r.t. the scope but takes substantially more time. Finally, pruning equivalences based on a relatively small but diverse suite of test valuations works similarly to dynamic pruning.

This paper makes the following contributions:

Problem: We are the first to study the problem of expression generation for relational algebra.

Optimizations: We introduce a suite of equivalence pruning rules for relational expressions to improve the efficacy of expression generation.

Experiments: We present an experimental evaluation based on problems derived from 12 Alloy models; the results show that RexGen with static pruning offers a promising approach for generating non-equivalent relational expressions.

2 Example

We next present an example model to motivate relational expression generation and introduce the basic concepts of our approach. Consider this small but illustrative Alloy model of directed trees, adapted from a recent paper [32]:

```
sig Node { edges: set Node }
pred Acyclic { no iden & ^edges }
pred Injective { edges.~edges in iden }
pred Connected { (Node -> Node) in ^(edges + ~edges) }
pred isDirectedTree { Acyclic and Injective and Connected }
run isDirectedTree for 4 Node
```

The model declares a set (called *signature* in Alloy) of nodes with a *field* called `edges` that is a binary relation of type `Node`$\times$`Node`. The keyword `set` declares an arbitrary relation; Alloy also has keywords `one` and `lone` to constrain the relation to be a total or partial function, respectively. The *predicate* (`pred`) is a named formula that can be invoked elsewhere. The conjunction of `Acyclic`, `Injective`, and `Connected` would precisely represent directed trees. The binary operator `&` is set intersection; `+` is set union; `in` is subset; `.` is relational join (and relational image); and `->` is Cartesian product. The (prefix) unary operator `^` is transitive closure, and $\sim$ is transpose; Alloy also has *reflexive* transitive closure (`*`). The keyword `iden` represents the identity relation. The formula `no E` for expression `E` constrains `E` to be the empty set. The `run` command runs a given formula, and presents an instance of the given formula if the formula is satisfiable. The scope of 4 instructs the analyzer to create an instance with at most 4 nodes.

To illustrate expression generation using our approach, consider the signature declaration in this model, which introduces one set (`Node`) and one binary relation (`edges`). Given those declarations, a user may want to generate various expressions, e.g., in synthesis or repair tasks. For example, many Alloy beginners write `pred Acyclic' { all n: Node | n !in n.^edges }` and may want to know if there is a semantically equivalent formula without any quantified variables (as in `Acyclic`). In that case, the user may want to systematically try `{UO E}` where `UO` represents any unary operator (`no`, `some`, `lone`, `one`) and `E` represents any valid expression, such that the formula `{UO E}` is equivalent to `Acyclic'`.

Assume we set the maximum size of any generated expression to 5, which suffices to generate even the largest relational expressions in this particular model. RexGen generates 581 expressions with no pruning, 438 with AC pruning (i.e., associativity and commutativity), 116 with static pruning, 105 with dynamic pruning, and 102 with modulo-instance pruning (for 14 tests). The generation time is largest for dynamic pruning, which uses Alloy analyzer to check each equivalence and takes 2.8 sec; in all other cases, no constraint solving is used, and the generation time is <1 sec. The following shows some of the equivalences discovered with dynamic pruning (where `univ` denotes the universe of discourse, which is equal to `Node` in the example model):

```
Node->Node = univ->univ                 | (~edges)&(^edges) = (~edges)&(*edges)
(~edges).Node = Node.edges              | *((^edges)-edges) = *((*edges)-edges)
edges.(Node.edges) = edges.Node         | ^(edges.(^edges)) = edges.(^edges)
```

To illustrate generation of larger expressions, consider size 7. RexGen generates 17109 expressions with no pruning, 11191 with AC pruning, 1464 with static pruning, 771 with dynamic pruning, and 691 with modulo-instance pruning (for 14 tests). The generation time for dynamic pruning increases to 82.3 sec, for modulo-instance pruning increases to 1.7 sec, and for the other techniques remains <1 sec. Thus, for this example, static pruning reduces the number of expressions by 86.9% over AC pruning while taking a similar amount of time; dynamic pruning reduces the number by 47.3% over static pruning but takes much longer due to many SAT calls. Moreover, modulo-instance pruning creates a similar number of expressions as dynamic pruning, which indicates the diversity of the tests, but takes less time due to not making SAT calls.

3 RexGen Framework

We next present our ***Re****lational* ***Ex****pression* ***Gen****erator* (RexGen) approach for generating non-equivalent relational expressions. We first describe the technique input and then the expression generation techniques.

3.1 Technique input

RexGen takes as input (1) a number of sets (signatures), relations (fields), and variables declared in an Alloy model (in the context in which the expressions should be generated), (2) a limit on the size of generated expressions, (3) optionally a target arity of expressions to generate, and (4) optionally a number of test valuations, i.e., values for the input sets and relations (but not for the bound variables). RexGen generates expressions using the following grammar:

$$\begin{aligned} \mathit{expr} &::= \mathit{expr}\ \mathit{binOp}\ \mathit{expr} \ \mid\ \mathit{expr}^{*} \ \mid\ \mathit{expr}^{+} \ \mid\ \mathit{expr}^{-1} \ \mid\ \mathit{terminal} \\ \mathit{binOp} &::= \cup \ \mid\ \cap \ \mid\ \setminus \ \mid\ \times \ \mid\ \bowtie \\ \mathit{terminal} &::= \mathit{set} \ \mid\ \mathit{relation} \ \mid\ \mathit{variable} \end{aligned}$$

The grammar captures a subset of syntactically possible Alloy expressions, which cover a large space of candidate expressions likely to be intended by Alloy users. For example, we do not consider rarely used Alloy operators such as domain restriction (`<:`). We use standard notation of relational algebra: $\cup$ is set union, $\cap$ is set intersection, $\setminus$ is set difference, $\times$ is Cartesian product, $\bowtie$ is the relational join; e^*, e^+, e^{-1} denote the reflexive transitive closure, transitive closure, and transpose of e, respectively. Additionally we use the empty set $\emptyset$, the universal set $univ$, and the identity $iden = \{(x, x)|x \in univ\}$.

To systematically generate expressions, RexGen limits: (1) the size of expressions and (2) the maximum arity of expressions. There are different ways to define expression size; we consider the number of AST nodes in the expression: $size(terminal) = 1, size(e_1\, binOp\, e_2) = size(e_1) + size(e_2) + 1, size(expr^{unOp}) = size(expr) + 1$.

3.2 Generating expressions

We next describe how RexGen enumerates expressions within the given limits. In the spirit of synthesis tools [3], enumeration works bottom-up, starting from *terminal* expressions (sets, relations, and variables given as inputs) and then iteratively combining smaller expressions to generate larger ones.

Our key contribution is pruning that aggressively removes expressions to increase the efficiency of the generation and/or reduce the number of generated expressions. The goal of pruning is to eliminate expressions that are semantically equivalent with previously generated expressions. Pruning has three modes: *static*, *dynamic*, and *modulo* pruning.

Expression generation algorithm. The generation algorithm maintains a list of expressions, $exprs[arity]$, indexed by the arity. The list maintains a total order among expressions of the same arity; we use $ind(e)$ to denote the index of the expression e in the list, and some pruning rules use this index.

The lists are instantiated with the terminal expressions (i.e., sets, relations, and variables declared in the model), based on their arity. The size of these expressions is 1. Then, until a limit is reached, the algorithm iteratively increases size and combines every operator and every combination of expressions of appropriate smaller sizes to generate expressions of the larger size. Each generated expression is then added to $exprs$ if it is (1) within the limits given for the generation, (2) well typed in Alloy, and (3) not pruned by the current pruning mode. Note that, by construction, expressions in $exprs$ are syntactically different. The rest of this section explains in detail well typedness and the three pruning modes.

Well typedness. RexGen tracks type information for generated expressions, typically using the default Alloy type system, which includes subset/subtyping and union types [16]. However, for some expressions, RexGen tracks a more precise type than the default type system. The main reason is the semantics of reflexive transitive closure (*). In Alloy, reflexive transitive closure is a superset of the identity relation for the union of all sets ($univ$) and thus has type $univ \times univ$. For example, if a model has two sets, $Node$ and $Value$, and a relation, $edges$, of type $Node \times Node$, then $edges^*$ is not of type $Node \times Node$ but $univ \times univ$, where $univ = Node \cup Value$. However, this type is too broad; it allows for arbitrary applications of other operators and makes expression generation intractable, producing expressions that are not intended in practical use.

For example, consider the expression $a^* + b$, where a has type $A \times A$. Intuitively, we want to allow only expressions of type $A \times A$ for b; however, we cannot track this precisely if we allow a^* to have type $univ \times univ$. On the other hand, we cannot consider a^* to have type $A \times A$ because that would make a^* a subset of $A \times A$, causing the static pruning to incorrectly prune expressions like $a^* + A \times A$. Therefore, RexGen conceptually uses a special type system to type intermediate generated expressions, but uses Alloy type for static pruning.

Static pruning. Static pruning removes expressions that are known to be semantically equivalent with other generated expressions. This pruning considers equivalence with respect to *all* possible valuations not only given test valuations. To prune equivalent expressions, we derive a comprehensive suite of equivalence rules specific to relational algebra. Other generation systems [36] use similar pruning rules for other domains, but our work is the first to provide rules specific to relational algebra.

Table 1 presents the static pruning rules of RexGen. The first column gives the pattern of equivalent expressions that the rule intends to eliminate. RexGen prunes the expression whose syntactic shape is the left-hand side of the equivalence. The second column specifies the condition for pruning. Note that almost all rules use only syntactic information or type (and arity) information for the involved expressions, which makes the rules easily checkable. An exception are a few rules that check the subset property between two sets/relations; because subset is a semantic property and not easily checkable, we approximate it conservatively, as shown in Table 2. Another exception is the rule for commutativity. To avoid generating both *a op b* and *b op a*, where *op* is a commutative operation, we use the total order defined for each arity by *exprs*: we prune the expression with $ind(a) > ind(b)$, where $ind(e)$ is the index of e in the list *exprs*.

Dynamic pruning. Dynamic pruning removes equivalent expressions by using the Alloy analyzer to check whether an expression is equivalent to another one already generated. Unlike static pruning, dynamic pruning considers (1) all signature/field constraints (e.g., that a relation must be a function) and (2) bound variables in the scope of the generated expression. To our knowledge, no previous work handles variables locally bound by a quantifier in the scope of the generated expression.

For a new expression, E, and a previously generated expression, E', RexGen creates a new Alloy model that includes all signature/field declarations from the RexGen input plus `check { all` v_1`:` D_1 `|...| all` v_n`:` D_n `|` E `=` E' `}`, where $v_1 \ldots v_n$ are variables used in the two expressions (except for sigs/fields from the model) and $D_1 \ldots D_n$ are their corresponding domains. For example, if E is `n.~edges` and E' is `Node.*edges`, then the equivalence checking command is `check { all n: Node | n.~edges = Node.*edges }`. This check is issued for every previously generated expression in *exprs* until either the new expression is found equivalent to some previously generated one, or the new expression is found not equivalent to any previously generated one and is thus added to *exprs*. Dynamic

Table 1. Static pruning rules

Equivalence (lhs = rhs)	Condition if needed; otherwise `true`
$a\ op\ (b\ op\ c) = (a\ op\ b)\ op\ c$	op associative
$a\ op\ b = b\ op\ a$	op commutative and $ind(a) > ind(b)$
$a \cup b = b$ and $b \cup a = b$	$[\![a]\!] \subseteq [\![b]\!]$
– Similar for $\cap$ and $\supseteq$	
$a \setminus b = \emptyset$	$[\![a]\!] \subseteq [\![b]\!]$
$a \cup b = c \cup b$	$\exists c. a \cong c \cup b$ or $a \cong b \cup c$ or $a \cong c \setminus b$
$a \cup b = b$	$\exists c. a \cong c \cap b$ or $a \cong b \cap c$ or $a \cong b \setminus c$
– Also symmetrically	– where $\cong$ is syntactic pattern matching
$(a\ op_1\ b)\ op_2\ (a\ op_1\ c) = a\ op_1\ (b\ op_2\ c)$	$op_1 \in \{\bowtie, \times, \cap\}, op_2 \in \{\cup, \cap\}$
– Similar for $(a\ op_1\ b)\ op_2\ (c\ op_1\ b)$	
$a^{-1}\ op\ b^{-1} = (a\ op\ b)^{-1}$	$op \in \{\cup, \cap, \setminus, \bowtie\}$
$\bigcup e_i = \bigcup_{i \neq j} e_i,$	$e_j \cong e_k$ for some $j \neq k$
– Similar for $\bigcap$	
$a \setminus (b \cup c) = (a \setminus b) \setminus c$	
$a \setminus (a \cap b) = a \setminus b$	
– Similar for $a \setminus (b \cap a)$	
$a \setminus (a \setminus b) = a \cap b$	
$a \setminus (b \setminus a) = a$	
$(a \cup b) \setminus a = b \setminus a$	
$(a\ op\ b) \setminus (a\ op\ c) = a\ op\ (b \setminus c)$	$op \in \{\times, \cap\}$
$(a \cap b) \setminus c = a \cap (b \setminus c)$	
$a \bowtie (a \times b) = b$	$card(a) \geq 1$
– Similar for $(b \times a) \bowtie a$	
$a \bowtie b^{-1} = b \bowtie a$	$arity(a) = 1$
$A \bowtie b^* = A$	$b : A \times A$
– Similar for $b^* \bowtie A$	– where $b : A \times A$ means that b has type $A \times A$
$A \bowtie b^+ = A \bowtie b$	$b : A \times A$
$b^+ \bowtie A = b \bowtie A$	$b : A \times A$
$b \bowtie b^* = b^+$	
– Similar for $b^* \bowtie b$	
$a^{*+} = a^*$	
– Similar for a^{+*}	
$a^{-1^{-1}} = a$	
$a^{*^{-1}} = a^{-1*}$	
$a^{+^{-1}} = a^{-1+}$	
$(a\ op\ b^{-1})^{-1} = a^{-1}\ op\ b$	$op \in \{\cup, \cap, \setminus, \bowtie\}$
$(a \times b)^+ = a \times b$	
$a \bowtie (b \times c) = (a \bowtie b) \times c$	$arity(a) + arity(b) > 2$
– Similar for $(a \times b) \bowtie c$	
$b^{-1} \bowtie a = a \bowtie b$	$arity(a) = 1$
$a^+ \bowtie a = a \bowtie a^+$	
$a^* \bowtie a^* = a^*$	
$a^* \bowtie a^+ = a^+$	
– Similar for $a^+ \bowtie a^*$	
$a^+ \bowtie a^+ = a \bowtie a^+$	
$(a \setminus b) \bowtie (b \times c) = \emptyset$	
– Also symmetrically	
$a \bowtie ((b \setminus a) \times c) = \emptyset$	
– Also symmetrically	
$A \bowtie (A \times b) = b$	$arity(A) = 1$
– Similar for $(b \times A) \bowtie A$	$b : B_1 \times ... \times A \times ... \times B_n$ for some $B_i = A$

Table 2. Syntactic approximation for $a \subseteq b$. $\cong$ means syntactic match.

1. $b \cong A$, $a : A$	5. $a \cong b \setminus c$
2. $a \cong b$	6. $a \cong c^+, b \cong c^*$
3. $b \cong a \cup c$ or $b \cong c \cup a$	7. $a \cong c \bowtie c \bowtie \ldots \bowtie c, b \cong c^+$ or $b \cong c^*$
4. $a \cong b \cap c$ or $a \cong c \cap b$	8. $a \cong c \times c$, $b \cong d^*$, a has cardinality 1, c has arity 1

pruning can be applied to all expressions for every arity or only expressions of the target arity.

Modulo pruning. Modulo pruning [54] removes equivalent expressions based on their values for the user-given valuations of the input test suite. Specifically, modulo pruning builds equivalence classes of expressions by grouping together all expressions that *evaluate* to the same value across all test valuations, and keeping only one expression per equivalence class.

Modulo pruning determines an expression's equivalence class without constraint *solving*, by utilizing the `Evaluator` feature of the Alloy Analyzer to perform constraint *checking*. The `Evaluator` takes as input an Alloy instance and an Alloy expression, and returns the concrete value of the expression for the given instance. For a new expression E, modulo pruning evaluates E for every test valuation in the suite, building a map of E's concrete values. If E contains any free variable(s), modulo pruning evaluates E for each element in the variable's domain, or more generally, for the cross product of domain elements if E contains multiple variables. If E's concrete-value map matches a previous expression, then E is pruned out; otherwise, E is kept. Modulo pruning only determines equivalence based on the user-given test suite, not guaranteeing equivalence across all instances in scope as dynamic pruning does.

4 Experimental evaluation

We next present our experimental evaluation of RexGen. We use 12 diverse Alloy models for evaluation (Section 4.1). We evaluate the number of expressions RexGen generates and the time it takes for each model under different settings (Section 4.2).

4.1 Evaluation models

We evaluate RexGen using 12 models comprised of a wide variety of example, educational, and "real-world" specifications. Address book (*addr*), Dijkstra mutual exclusion algorithm (*dijkstra*), farmer crossing-river puzzle (*farmer*), Halmos handshake problem (*hshake*), and genealogy (*gene*) are from the Alloy's distribution examples. Bad employee (*bempl*), colored tree (*ctree*), directed tree (*dtree*), and grade book (*grade*) are Alloy translations of access-control specifications used to evaluate existing scenario-finding work [32,43]. Binary tree (*btree*) constrains the graph to be a binary tree. Propositional resolution (*resfm*) is from Torlak et al. [52]. Singly linked list (*sll*) models acyclic lists.

Table 3 shows the basic information of these models. *Model* is the name. *#AST* is the number of AST nodes in each model. *#Sig* is the number of signatures declared in each model. *#Rel* is the number of relations declared in

Table 3. Basic information of models used to evaluate RexGen

Model	**#AST**	**#Sig**	**#Rel**	**#Var**	**#PrimVar**	**#Test**
addr	114	4	2	8	45	14
bempl	46	6	3	11	38	14
btree	53	2	2	6	24	14
ctree	71	4	2	8	18	14
dijkstra	385	3	1	10	57	14
dtree	49	1	1	2	12	14
farmer	169	6	3	14	24	14
gene	139	5	2	8	20	14
grade	64	5	4	11	48	14
hshake	127	3	2	6	19	14
resfm	285	8	7	19	101	14
sll	33	2	2	5	15	14

each model. For each model, we find all identifiers in scope, including signatures, relations, and bound variables, for the *largest* expression (w.r.t. our measure of size). *#Var* is the number of all identifiers in scope to generate expressions. In our experiment, we first find the expression with the largest size in each model and then use all sigs, relations, and variables in the scope of that expression to generate more expressions. *#PrimVar* is the number of primary variables when we run an empty command (`run {}`) without test-specific constraints; it represents the basic complexity of signature declarations and constraints that always hold in each model. *#Test* is the number of tests; we use the same number of tests for each model so that the results do not depend on the number of tests. We chose the number of tests based on the *sll* model, where we create tests such that modulo pruning generates the same number of expressions of size 4 as dynamic pruning for this model. We iteratively add tests until modulo pruning and dynamic pruning create the same set of expressions. In the end, we obtain 14 tests for *sll* and use the same number of tests for other models.

Our experiments are performed on a MacBook Pro running OS X El Capitan with 2.5 GHz Intel Core i7-4870HQ and 16GB of RAM.

4.2 RexGen results

Table 4 shows the performance of RexGen across different expression pruning environments: *No Pr.* uses no pruning rules, *AC Pr.* uses just associativity and commutativity pruning rules, *Static Pr.* uses all static pruning rules, *Dynamic Pr.* uses dynamic pruning, and *Modulo Pr.* uses modulo-instance pruning. Note that both dynamic pruning and modulo-instance pruning are applied on expressions after they are pruned by static pruning. Column *Problem* shows the Alloy model and the corresponding size used for generation. For each pruning environment, *#expr* shows the number of expressions generated and *time* shows the time duration in milliseconds to generate all expressions, with a time-out of one hour. The number of generated expressions shown in the table is for expressions of all arities up to 3.

Expression generation using *No Pr.*, *AC Pr.*, and *Static Pr.* is fast, taking at most 7.9 seconds (*farmer* and size 7 using *No Pr.*), but frequently finishing in

Table 4. RexGen performance. Times are in ms. ⊥ indicates a timeout (>1 hour).

Problem		**No Pr.**		**AC Pr.**		**Static Pr.**		**Dynamic Pr.**		**Modulo Pr.**	
		#expr	**time**	**#expr**	**time**	**#expr**	**time**	**#expr**	**time**	**#expr**	**time**
addr	4	231	2	199	4	129	25	118	1259	108	279
	5	3984	18	2374	19	1335	56	823	15869	600	1396
	6	7913	27	5563	29	2034	64	1193	19359	900	1879
	7	139971	204	65346	131	24839	189	7116	635296	3546	7902
bempl	4	427	5	377	6	261	29	246	1939	237	343
	5	7027	27	4369	25	2463	64	1708	25098	1424	2999
	6	15396	50	11144	41	4096	80	2588	43840	2198	3814
	7	254843	363	128706	274	47747	296	15363	1555983	10309	29174
btree	4	415	4	355	6	223	29	215	6247	196	484
	5	3264	18	2391	18	1032	51	915	62153	740	1920
	6	17956	42	12919	41	4553	93	3424	999892	2227	6221
	7	139882	204	88578	148	25031	195	⊥	⊥	8505	26140
ctree	4	369	4	327	6	202	27	185	2773	144	754
	5	4625	21	3031	21	1446	59	996	28674	737	5282
	6	14315	37	10707	38	3473	79	2143	192314	1169	9584
	7	168181	221	93805	175	27660	213	⊥	⊥	5530	60968
dijkstra	4	287	2	251	4	140	26	135	2235	133	264
	5	4661	19	2763	20	1397	53	1097	20544	1069	2185
	6	9939	30	7159	39	2175	60	1637	36446	1552	3083
	7	138703	213	65991	139	17976	180	7007	670275	5704	17820
dtree	4	111	1	95	3	40	21	38	680	37	144
	5	581	4	438	6	116	30	105	2809	102	401
	6	2957	15	2130	14	376	39	250	11247	234	841
	7	17109	40	11191	34	1464	61	771	82268	691	1686
farmer	4	1077	8	939	8	654	39	619	28327	454	695
	5	41007	73	24322	53	16969	141	⊥	⊥	5116	8992
	6	96607	140	68468	124	33097	215	⊥	⊥	9247	22555
	7	3666499	7942	1661501	4581	923952	3985	⊥	⊥	80553	2156722
gene	4	641	5	551	6	376	32	348	10853	242	916
	5	12055	31	7653	29	4597	83	3228	632913	1675	8614
	6	42897	76	30703	64	14621	145	⊥	⊥	4324	20490
	7	763031	1998	393015	1150	177920	665	⊥	⊥	26222	326804
grade	4	421	4	373	6	267	30	244	2570	229	447
	5	6533	25	4168	25	2342	65	1496	28995	1105	2450
	6	14930	45	11033	42	4141	89	2321	52542	1740	3446
	7	234482	373	122012	258	45312	311	13166	1858565	7300	19416
hshake	4	471	3	403	5	260	31	244	8543	173	1131
	5	5625	20	3805	22	1936	61	1505	164478	1020	8180
	6	25523	51	18319	46	7640	112	5378	2570827	3031	21775
	7	286661	355	163874	247	58505	318	⊥	⊥	16149	222019
resfm	4	1030	8	940	12	652	38	625	10051	510	767
	5	18692	50	12250	42	7337	99	5705	336197	3626	5634
	6	47128	111	35984	94	13384	146	9406	938031	5026	9076
	7	822434	2107	449935	653	175337	845	⊥	⊥	24997	181425
sll	4	209	2	183	4	104	25	98	1468	98	283
	5	1549	10	1100	13	397	40	331	6868	330	987
	6	6267	25	4694	25	1203	58	808	37988	803	2593
	7	45527	86	28622	64	5463	95	2712	429769	2671	9391

under a second. Accordingly, both *AC Pr.* and *Static Pr.* have negligible overhead. However, the number of expressions generated can vary greatly, as seen in Table 4. *No Pr.* generates all possible expressions and provides a means of measuring the effectiveness of different pruning environments. Compared to *No Pr.*, *AC Pr.* reduces the number of expressions generated by 8.7–54.7%, while *Static Pr.* reduces the number of expressions generated by 36.6–91.4%. Compared directly, *Static Pr.* generates 28.4–86.9% fewer expressions than *AC Pr.*. In other words, *Static Pr.*'s additional pruning rules highlight that associativity and commutativity are not strong enough to prune relational expressions on their own. Moreover, Table 4 shows that the pruning rules for *AC Pr.* and *Static Pr.* reduce the space of possible expressions by a large enough degree that both techniques often finish faster than *No Pr.*, despite the time they spend on applying equivalence rules to check expressions. Although *Static Pr.* has 40 more rules than *AC Pr.*, the difference in runtime between *AC Pr.* and *Static Pr.* is often less than a second. Therefore, *Static Pr.*'s rules are inexpensive to run but effective at reducing the number of generated expressions.

We can analyze expressions to prune out more equivalences. *Dynamic Pr.* further prunes expressions generated by *Static Pr.*; *Dynamic Pr.* is motivated by using Alloy to find all equivalences (within a given scope), thus capturing equivalences which cannot be captured by generic static pruning rules. As expected, *Dynamic Pr.* reduces the number of expressions from *Static Pr.*, by 3.6–71.4%. *Dynamic Pr.* gives the minimum number of non-equivalent expressions for each model, showing the lower bound of what *Static Pr.* could achieve.

Modulo Pr. also filters expressions generated by *Static Pr.*; specifically, *Modulo Pr.* reduces the expressions from *Static Pr.* by 5.0–91.3%. *Dynamic Pr.* can be viewed as *Modulo Pr.* if the input test suite covered all instances in scope. However, since we use only 14 tests per model for *Modulo Pr.*, *Modulo Pr.* may even prune expressions that are semantically non-equivalent up to a given scope but equivalent over all 14 tests. For example, *Modulo Pr.* prunes 10 more size 4 expressions and 223 more size 5 expressions for *addr* compared to *Dynamic Pr.*. Therefore, as expected, *Modulo Pr.* can reduce the number of generated expressions compared to *Dynamic Pr.*, by as much as 50.2% (*addr*), or *Modulo Pr.* can generate the same number of expressions (*sll* and size 4) but 5.2× faster. The trade-off is that, while *Dynamic Pr.* is guaranteed to not prune expressions that are semantically non-equivalent within a given scope, it is slower than *Modulo Pr.*; *Dynamic Pr.* times out on 7 different problems, while *Modulo Pr.* frequently finishes in under a minute, with the longest runtime being 2156.7 seconds. While *Modulo Pr.* provides a practical, lighter-weight alternative to *Dynamic Pr.*, *Modulo Pr.* still has a high overhead over *Static Pr.*. For instance, for *farmer* and size 7, *Static Pr.* can generate expressions in 4.0 seconds, while *Modulo Pr.* needs 2156.7 seconds to finish.

In our experiment, applying *Dynamic Pr.* or *Modulo Pr.* on expressions generated with *No Pr.* or *AC Pr.* takes significantly longer. *Static Pr.*'s ability to significantly reduce the number of generated expressions, with a negligible overhead, makes *Static Pr.* the recommended approach for relational expression generation (even when considering more advanced pruning techniques like *Dynamic Pr.* or *Modulo Pr.*). To check that our static pruning rules are correct, we ran dynamic pruning on expressions generated using *AC Pr.* and *Static Pr.*: we found that the numbers of non-equivalent expressions generated after dynamic pruning for both *AC Pr.* and *Static Pr.* are exactly the same, which indicates that *Static Pr.* does not incorrectly prune any non-equivalent expression.

5 Related work

Enumeration algorithms include bottom-up enumeration [3,54], used by RexGen, and top-down enumeration [6]. EuSolver [3] has been one of the most prominent solvers in Syntax-Guided Synthesis (SyGuS) competitions. FlashMeta [38] uses version-space algebra to concisely represent a large number of programs. Neither EuSolver nor FlashMeta focus on relational expressions, which can generate a large number of equivalent expressions. Our work proposes a number of pruning rules that substantially reduce the number of equivalent expressions, thus providing basis for practical synthesis with relational expressions.
Search space pruning of expression generation is important because search spaces for any realistic programming language quickly become intractable. Pruning techniques include indistinguishability of expressions modulo a set of inputs [3,54] and partial evaluation of incomplete expressions [6]. Knowledge about operator properties has also been used to explore equivalent expressions, either after expression generation [36] or by applying an automated transformation to the grammar which represents candidate programs [23]. However, most techniques have only been explored in the domains of integers, booleans, and abstract data types, all of which have less comprehensive sets of equivalence rules than our work with the domains of sets and relations.
Applications of expression generation are quite common. For example, program synthesis has attracted attention for a few decades [28], and researchers have applied it in a variety of domains [5,6,8,12,22,27]. Program sketching [46] is another example, which demonstrated the opportunities to apply modern solver technology to the synthesis problem, and introduced the counter-example guided inductive synthesis paradigm to program synthesis. Sketch requires the user to provide generators of expressions for expression holes [2, 14, 19, 45]. While most work on sketching is in the context of synthesis, SketchRep [13] applies sketching to the problem of program repair [10, 20, 24, 40, 57], i.e., correcting faulty lines of code. Synthesis from examples, the inspiration behind test valuations, has also been extensively studied [1, 35]. Notably, synthesis from examples has been successfully employed in commercial products [11]. EdSketch [14] introduced an optimized backtracking search for completing Java sketches using test executions for pruning. SketchFix [15] used EdSketch as the backend synthesis engine for program repair. EdSynth [58] builds on EdSketch and synthesizes method sequences for given sketches that may contain conditional branches. SyPet [5] introduced a novel use of Petri nets in synthesizing straightline sequences of method invocations for complex APIs using tests. The key enabler of all of the

above applications is efficient expression generation; ours is the first work that addresses generation for relational algebra.

Alloy is a well studied lightweight modeling approach that has been applied in various domains, including software design [29, 30], networking [41], and security [26, 34]. This paper is the first to study expression generation for Alloy and more generally for relational algebra. Our work leverages the AUnit [48, 51] approach for writing tests for Alloy models. Various approaches assist Alloy users to build their models correctly, e.g., by improving scenario exploration [32, 33], supporting state modeling [7,17,18,31,49], highlighting UNSAT cores [44,52,53], and creating tests [50,55]. RexGen provides a novel basis of a synthesis or sketching engine for Alloy in particular and relational logic in general [47, 56].

6 Conclusions

We introduced RexGen, the first generator for non-equivalent relational expressions. We presented a set of equivalence rules for relational expressions, used them for pruning in our generator, embodied the generator based on the Alloy tool-set, and presented an experimental evaluation of the effectiveness of our non-equivalent generation for a variety of problems with relational constraints. RexGen provides the key step to address the broader problems of synthesis and repair of declarative models in Alloy. Our companion paper on ASketch [56] shows how to use the generated expressions to synthesize Alloy models from sketches. We hope our work inspires the development of a broader tool-set to support software models and eventually leads to more reliable software systems.

Acknowledgements

We thank Viktor Kuncak for his comments on this work. Manos Koukoutos is supported in part by the European Research Council (ERC) project "Implicit Programming". This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-1409423, CCF-1421503, CNS-1646305, CCF-1718903, and CNS-1740916.

References

1. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV (2013)
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD (2013)
3. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: TACAS (2017)
4. Dennis, G., Chang, F.S., Jackson, D.: Modular verification of code with SAT. In: ISSTA (2006)
5. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs. In: POPL (2017)
6. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: PLDI (2015)

7. Frias, M.F., Galeotti, J.P., Pombo, C.G.L., Aguirre, N.M.: DynAlloy: Upgrading Alloy with actions. In: ICSE (2005)
8. Galenson, J., Reames, P., Bodik, R., Hartmann, B., Sen, K.: CodeHint: Dynamic and interactive synthesis of code snippets. In: ICSE (2014)
9. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: efficient SAT-based bounded verification using symmetry breaking and tight bounds. TSE (2013)
10. Gopinath, D., Malik, M.Z., Khurshid, S.: Specification-based program repair using SAT. In: TACAS (2011)
11. Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S.H., Schmid, U., Zorn, B.: Inductive programming meets the real world. CACM (2015)
12. Gvero, T., Kuncak, V., Piskac, R.: Interactive synthesis of code snippets. In: CAV (2011)
13. Hua, J., Khurshid, S.: A sketching-based approach for debugging using test cases. In: ATVA (2016)
14. Hua, J., Khurshid, S.: EdSketch: Execution-driven sketching for Java. In: SPIN (2017)
15. Hua, J., Zhang, M., Wang, K., Khurshid, S.: Towards practical program repair with on-demand candidate generation. In: ICSE (2018)
16. Jackson, D.: Alloy: A lightweight object modelling notation. TSE (2002)
17. Jackson, D., Fekete, A.: Lightweight analysis of object interactions. In: TACS (2001)
18. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA (2000)
19. Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: JSketch: Sketching for Java. In: FSE (2015)
20. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: CAV (2005)
21. Kang, E., Milicevic, A., Jackson, D.: Multi-representational security analysis. In: FSE (2016)
22. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA (2013)
23. Koukoutos, M., Kneuss, E., Kuncak, V.: An update on deductive synthesis and repair in the leon tool. In: SYNT Workshop (2016)
24. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: FSE (2015)
25. Maier, D.: Theory of Relational Databases. Computer Science Pr (1983)
26. Maldonado-Lopez, F.A., Chavarriaga, J., Donoso, Y.: Detecting Network Policy Conflicts Using Alloy (2014)
27. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: Helping to navigate the API jungle. PLDI (2005)
28. Manna, Z., Waldinger, R.: Toward automatic program synthesis. CACM **14**(3) (1971)
29. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: MODELS (2011)
30. Maoz, S., Ringert, J.O., Rumpe, B.: CDDiff: Semantic differencing for class diagrams. In: ECOOP (2011)
31. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: ASE (2001)
32. Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": Enriching scenario exploration with provenance. In: FSE (2017)
33. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: ICSE (2013)

34. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA (2010)
35. Pei, Y., Furia, C.A., Nordio, M., Meyer, B.: Automated program repair in an integrated development environment. In: ICSE (2015)
36. Perelman, D., Gulwani, S., Grossman, D., Provost, P.: Test-driven synthesis. PLDI (2014)
37. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: PLDI (2016)
38. Polozov, O., Gulwani, S.: FlashMeta: A framework for inductive program synthesis. In: OOPSLA (2015)
39. Richters, M., Gogolla, M.: Ocl: Syntax, semantics, and tools. In: Object Modeling with the OCL, The Rationale Behind the Object Constraint Language (2002)
40. Rothenberg, B., Grumberg, O.: Sound and complete mutation-based program repair. In: FM (2016)
41. Ruchansky, N., Proserpio, D.: A (not) NICE way to verify the Openflow switch specification: Formal modelling of the Openflow switch using Alloy. SIGCOMM (2013)
42. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, The (2nd Edition) (2004)
43. Saghafi, S., Danas, R., Dougherty, D.J.: Exploring Theories with a Model-Finding Assistant (2015)
44. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: ASE (2003)
45. Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: FSE (2011)
46. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
47. Sullivan, A.: Automated Testing and Sketching of Alloy Models. Ph.D. thesis, University of Texas at Austin (2017)
48. Sullivan, A., Wang, K., Khurshid, S.: AUnit: A test automation tool for Alloy. In: ICST (2018)
49. Sullivan, A., Wang, K., Khurshid, S., Marinov, D.: Evaluating state modeling techniques in alloy. In: SQAMIA (2017)
50. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST (2017)
51. Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for Alloy. In: SPIN (2014)
52. Torlak, E., Chang, F.S.H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: FM (2008)
53. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS (2007)
54. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: Specifying protocols with concolic snippets. In: PLDI (2013)
55. Wang, K., Sullivan, A., Khurshid, S.: MuAlloy: A Mutation Testing Framework for Alloy. In: ICSE (2018)
56. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: Solver-based sketching Alloy models using test valuations. In: ABZ (2018)
57. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE (2009)
58. Yang, Z., Hua, J., Wang, K., Khurshid, S.: Test execution driven synthesis of API sequences with conditionals and loops. In: ICST (2018)