Scalable Relational Analysis via Relational Bound Propagation

Clay Stevens
cdsteven@iastate.edu
Iowa State University
Department of Computer Science
Ames, Iowa, USA

Hamid Bagheri
bagheri@unl.edu
University of Nebraska-Lincoln
School of Computing
Lincoln, Nebraska, USA

ABSTRACT

Bounded formal analysis techniques (such as bounded model checking) are incredibly powerful tools for today's software engineers. However, such techniques often suffer from scalability challenges when applied to large-scale, real-world systems. It can be very difficult to ensure the bounds are set properly, which can have a profound impact on the performance and scalability of any bounded formal analysis. In this paper, we propose a novel approach—relational bound propagation—which leverages the semantics of the underlying relational logic formula encoded by the specification to automatically tighten the bounds for any relational specification. Our approach applies two sets of semantic rules to propagate the bounds on the relations via the abstract syntax tree of the formula, first upward to higher-level expressions on those relations then downward from those higher-level expressions to the relations. Thus, relational bound propagation can reduce the number of variables examined by the analysis and decrease the cost of performing the analysis. This paper presents formal definitions of these rules, all of which have been rigorously proven. We realize our approach in an accompanying tool, Propter, and present experimental results using Propter that test the efficacy of relational bound propagation to decrease the cost of relational bounded model checking. Our results demonstrate that relational bound propagation reduces the number of primary variables in 63.58% of tested specifications by an average of 30.68% (N=519) and decreases the analysis time for the subject specifications by an average of 49.30%. For large-scale, real-world specifications, Propter was able to reduce total analysis time by an average of 68.14% (N=25) while introducing comparatively little overhead (6.14% baseline analysis time).

KEYWORDS

formal methods, bounded model checking, bound tightening

ACM Reference Format:
Clay Stevens and Hamid Bagheri. 2024. Scalable Relational Analysis via Relational Bound Propagation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639171

1 INTRODUCTION

Formal analysis of software systems has long helped software engineers develop more secure, dependable, and efficient software. Using formal techniques, engineers first create a formal *specification* of their system, then analyze the specification to find models of the system that either satisfy or violate formal, logical properties of interest. This allows developers to (e.g.) prove that their systems are safe from threats [1, 2, 5, 32, 35]; synthesize and compare different system designs [6–9, 14, 17]; ensure dependability in safety-critical cyber-physical systems [3, 25]; and develop efficient self-adaptive systems [27, 36]. While advances in the state-of-the-art in recent years have lead to wider adoption in the industry [15, 28, 31, 39], formal analysis techniques still come with a steep cost; as the size of a software system (and its specification) grows, the processing required to analyze the system grows exponentially along with it. Many large-scale, real-world applications which would otherwise benefit from these techniques are thus rendered intractable for formal analysis, as the time required to perform the analysis outweighs the benefit. Recent advances in the underlying solvers coupled with *bounded* formal analysis techniques (such as *bounded model checking*) have greatly increased the scope of problems that can be solved with formal analysis. By placing a limit (or *bound*) on the scope of the analysis, these techniques sacrifice the global completeness of their analysis in exchange for greatly improved scalability. Even so, scalability remains a challenge for these improved techniques.

In particular, bounded techniques such as bounded model checking are sound and complete *up to the specified bounds*. Thus, it is in the interest of the analyzer to ensure that the bounds for the analysis are properly set. This can be particularly difficult for *relational* model checking (e.g., with Alloy), where the bounds must be specified for each relation in the specification. Correctly defining the bounds can be an arduous and arcane task, requiring a great deal of expertise in both the domain described and in the specification language itself. If the bounds are too tight (representing an *under-approximation*), the analysis may be faster, but relevant models or counterexamples of the specification may be missed. If the bounds are too loose (*over-approximation*), spurious models may be found and the analyzer will do unnecessary work.

Researchers have recently proposed a variety of techniques to use bound tightening to improve bounded relational model checking. These techniques have shown great promise in improving the scalability of formal analysis, but are limited in their applicability. In general, these state-of-the-art techniques leverage information gleaned from the domain of the system under analysis to tighten the bounds in very specific ways that either apply only to that domain [10, 16], require expensive processing [4], or require analysis-specific modifications [37].

This paper proposes a new approach to the problem of bound tightening that is not contingent upon the domain of the system. Our approach—*relational bound propagation*—leverages the semantics of the underlying relational logic formula encoded by the specification itself. By traversing the abstract syntax tree of the formula, our approach first applies a set of semantic rules to propagate the bounds on the *relations* of a relational model checking specification to the higher-level *expressions* applied to those relations. We then compare the bounds of those higher-level expressions in light of the logical constraints expressed in the formula, and use the result of those comparisons to tighten the bounds on the relations without sacrificing the completeness of the original bounds. Thus, relational bound propagation can exponentially reduce the size of the search space examined by the analysis for *any* relational specification and improve the scalability of bounded relational model checking. We realize our approach in a custom Java tool called Propter, which implements all the algorithms and propagation rules described in Section 4. We then used this tool to conduct an experimental evaluation of relational bound propagation. Our results show that Propter reduces both the size of the relational bounded model checking problem and the total analysis time for our subject specifications, by an average of 30.68% and 49.30%, respectively.

In summary, our contributions in this paper include:

- A novel approach, *relational bound propagation*, to automatically and validly tighten relational bounds;
- Formal definitions and proofs of the upward and downward *propagation rules* which drive our approach;
- Propter, an implementation of our approach which we make available to the community [38]; and
- Experimental results demonstrating Propter's ability to reduce both the size of the problems solved by relational bounded model checking and the analysis time required.

2 ILLUSTRATIVE EXAMPLE

The benefits of relational bound propagation can be described with a simple example, drawn from the domain of software architecture. Software architects must design complex software systems that satisfy a variety of constraints to ensure their system (a) meets its functional requirements and (b) maintains an appropriate level of security, performance, etc. Such constraints can be difficult to analyze manually, leading many software architects to use formal methods to prove their designs meet the constraints. Relational model checkers are especially valuable, as software architectures can be easily modeled with relational specifications [13, 18, 20, 22].

Figure 1 depicts a portion of the logical view of the software architecture for a service-oriented ecommerce system. The overall architecture is a *closed* four-tier architecture, meaning that each component may only invoke functionality from services in the layer directly below its own layer (e.g., client-layer services may only invoke logic-layer services). The system includes two client services—an app used by consumers to make orders and a website (viewed via a browser) which can be used to place orders as well as to manage vendor inventories. The logical view of this architecture can be easily represented in Alloy, as shown in Figure 2.

To analyze the properties of the architecture, the Alloy Analyzer would first translate the specification into a relational bounded

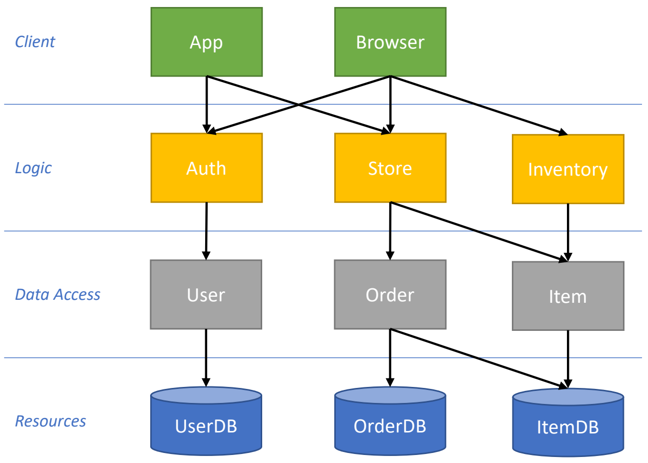


Figure 1: Example system architecture for a service-oriented ecommerce system (logical view).

```
// abstract sigs to represent the layers
abstract sig Component
   { calls: set Component }
abstract sig Client, Logic, Data, Resource
   extends Component {}
// client layer
one sig App extends Client {}
   { calls = Auth + Store }
one sig Browser extends Client {}
   { calls = Auth + Store + Inventory }
// logic layer
one sig Auth extends Logic {}
   { calls = UserAccess }
//... more service definitions ...
// data storage resources
one sig UserDB, OrderDB, ItemDB
   extends Resource {}
// closed architecture constraint
fact { (no calls.Client)
   and (Client.calls in Logic)
      and (Logic.calls in Data)
         and (Data.calls in Resource)
            and (no Resource.calls) }
// ... other constraints ...
```

Figure 2: Excerpt of Alloy specification for the example architecture (logical view).

model checking problem $\langle \mathcal{U}, \mathcal{R}, \mathcal{B}, \mathcal{F} \rangle$ where $\mathcal{U}$ is a set of undifferentiated atoms representing the universe; $\mathcal{R}$ is a set of relations defined over the atoms in $\mathcal{U}$; $\mathcal{B}$ is a set of lower and upper bounds defining which tuples *must* be assigned to each relation and which *may* be assigned, respectively; and $\mathcal{F}$ is a first-order logical formula in which the relations in $\mathcal{R}$ appear as free variables. Each top-level signature (denoted by "sig") in the Alloy specification corresponds to a unary relation in $\mathcal{R}$, the bounds of which are determined based on the scopes in the specification; the upper bounds of these unary relations collectively partition $\mathcal{U}$. Each field of those signatures corresponds to an n-ary relation in $\mathcal{R}$, the domains of which are typically drawn from the unary relations corresponding to the signatures. For most relations, the model finder defaults the lower bound to the empty set, and the upper bound to the Cartesian product of the upper bound of the domains of the relation. For example, for relation "calls" the default upper bound would include *all possible pairs* of atoms from the upper bound of "Component".

In the case of "calls" and other similarly defined relations, the default upper bound is much looser than would be required; as shown in Figure 1 and Figure 2, the assignments to those relations are already explicitly defined as part of the specification. The key insight for this approach is that those definitions can be discovered *before* running the analysis by examining the relational formula, $\mathcal{F}$, with respect to the bounds specified in $\mathcal{B}$. For example, Line 7 in Figure 2 can be translated to the following relational formula, where . signifies a relational join operation and ∪ is union:

$$\texttt{App.calls} = \texttt{Auth} \cup \texttt{Store} \tag{1}$$

Figure 3 provides an overview of how the relational bounds might propagate for the upper bounds of this subformula (a similar process is simultaneously applied for lower bounds). First, Figure 3a depicts *upward* propagation, where bounds are computed for each relational expression (e.g., ∪) based on the bounds of their child expressions. The upper bounds for `App`, `Auth`, and `Store` are all tightly specified (due to the "one" qualifier on the signature), each comprising exactly one atom. The initial upper bound for `calls` is the default upper bound described above (i.e., `Component` × `Component`). Those relation bounds are propagated up the tree by applying the upward propagation rules defined in Section 4.1.1, defining upper bounds for the nodes representing the results of applying each relational operator. Once the process encounters a *logical* operator (i.e., =), the bounds then can be propagated *downward*, expanding the lower bound and shrinking the upper bound as defined by the downward propagation rules in Section 4.1.2. Figure 3b depicts the downward propagation step, where the equality comparison at the top of the tree propagates the intersection of the bounds of its child nodes to each child. This results in removing tuples from the upper bound of the `calls` relation based on the relational join with `App`; any tuple in the upper bound of `calls` that begins with `App` but ends with an atom *other* than `Auth` or `Store` can safely be removed. Once the downward propagation reaches the leaves of the formula (i.e., the relations themselves), the possible changes to each relation's bounds are resolved based on the logical operators joining each clause and the bounds of the relations are updated. The process—both upward and downward—is repeated until no more changes are made to the bounds of any of the relations. In the example shown in Figure 3, the end result is that the upper bound of `calls` can be tightened, exponentially decreasing the size of the search space for the solver.

3 BACKGROUND: RELATIONAL BOUNDED MODEL CHECKING

Bounded model checking—as a general approach—has long been a key method of formally analyzing software systems. In general, the approach starts with some formal specification of a system or a problem (often a finite state machine), a set of formally-defined properties to check against that specification, and a *bound*—classically, an integer value that limits the number of transitions explored by the state machine. As the bound ensures the state space is finite, these can collectively be represented as a propositional satisfiability problem and solved using a SAT or SMT solver.

In a *relational* bounded model checking problem $\mathcal{P}$, the problem is specified as: (a) a universe $\mathcal{U}$ of undifferentiated atoms; (b) a set of *relations* $\mathcal{R}$ defined over those atoms; (c) a set of *bounds* $\mathcal{B}$ defining the tuples which can be assigned to those relations; and (d) a relational or first-order logic (RL) formula $\mathcal{F}$ in which the relations in $\mathcal{R}$ appear as free variables. The solver attempts to find an assignment of tuples to each relation in $\mathcal{R}$ which satisfies both $\mathcal{F}$ and the bounds defined in $\mathcal{B}$. Any such satisfying assignment m is a *model* of the problem, denoted $m \models \mathcal{P}$.

For each relation $r \in \mathcal{R}$, $\mathcal{B}$ contains a *lower* and a *upper* bound, expressed as the set the tuples that *must* be assigned to r in any satisfying model and the set of tuples that *may* be assigned to r in any satisfying model, respectively. This can be concisely expressed by borrowing some expressions from modal logics (such as the weak logic **K** [12]), taking $\Box\phi$ to mean the proposition ϕ is "necessary" (i.e., true in *all* satisfying models) and $\Diamond\phi$ to mean ϕ is "possible" in any satisfying model (i.e., $\neg\Box\neg\phi$, or it is true in *at least one* satisfying model). More formally, the notions of necessity and possibility can be defined as follows (where ϕ_m indicates ϕ is true under tuple assignment m):

Definition 1 (RL Necessity). *RL formula ϕ is said to be* necessary *if it is true under all tuple assignments m that model problem $\mathcal{P}$:*

$$\Box\phi \equiv \forall m\,(m \models \mathcal{P} \implies \phi_m)$$

Definition 2 (RL Possibility). *RL formula ϕ is said to be* possible *if it is true for at least one assignment m that models problem $\mathcal{P}$:*

$$\Diamond\phi \equiv \exists m\,(m \models \mathcal{P} \land \phi_m)$$

Using this notation, the upper bound and lower bound of a relation r are defined as follows (where x is a tuple of atoms from the same universe and with the same arity as relation r):

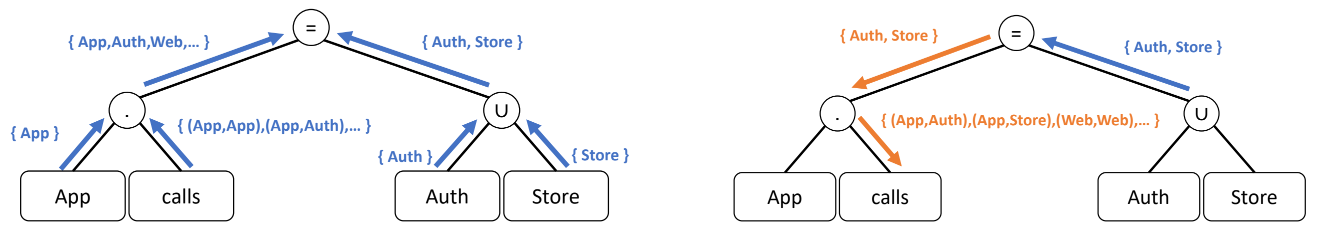
Definition 3 (Lower Bound). *The* lower bound *of relation r contains all tuples that* must *be assigned to r in any satisfying model of $\mathcal{P}$:*

$$\lfloor r \rfloor \equiv \{\, x \mid \Box(x \in r)\}$$

Definition 4 (Upper Bound). *The* upper bound *of relation r contains all tuples that* may *be assigned to r in a satisfying model of $\mathcal{P}$:*

$$\lceil r \rceil \equiv \{\, x \mid \Diamond(x \in r)\}$$

The relational formula $\mathcal{F}$ can be specified using the standard language of first-order logic with quantifiers combined with the language of set theory—i.e., union and intersection of sets, containment of tuples within sets, etc. The approach described here focuses on such formulas as can be expressed in Alloy [20]. These formulas generally comprise two types of statements—*expressions*, which define a set of tuples; and *formulas* which are Boolean-valued constraints on one or more sub-expressions or sub-formulas. Those formulas that compare or constrain expressions are termed *elementary* formulas, and those that constrain other formulas are *composite*



(a) Upward propagation. **(b)** Downward propagation.

Figure 3: Relational Bound Propagation Overview. Shows propagation of upper bounds in a portion of a specification describing a logical architecture. In (a), bounds from $\mathcal{B}$ are propagated up the AST of formula $\mathcal{F}$ from the relations in $\mathcal{R}$ according to the rules in Section 4.1.1 until the traversal encounters a set containment operator (=). In (b), the bounds on the operator node's children are tightened and propagated down the tree to the relations, possibly tightening bounds of one or more relations.

formulas (following the terminology in Fig. 1 of [40]). Our approach considers the usual set-theoretic operators (union, intersection, etc.) along with closure and relational joins.

Once specified, the relational bounded model checking problem $\mathcal{P}$ is converted into a purely propositional formula by computing the set of possible tuples that can be assigned to each relation r—given by $\lceil r \rceil \setminus \lfloor r \rfloor$—and creating a Boolean variable representing whether each such tuple is a member of the corresponding relation. These variables are called *primary variables*, and the assignment of truth values to the primary variables defines each possible model of $\mathcal{P}$. Satisfying models can therefore be found by translating the formula $\mathcal{F}$ into a format which can be interpreted by (e.g.) a SAT solver and having the solver compute satisfying assignments (if any) of truth values to the primary variables. Consequently, the solution space searched by the solver scales exponentially in the number of primary variables, so a reduction in that number will have a great impact on the size of the search space.

4 APPROACH

This paper introduces *relational bound propagation*, which leverages information encoded in the formula of a relational bounded model checking problem to decrease the size of the problem passed to the underlying solver by reducing the number of primary variables. We do so by extracting information encoded in the constraints in $\mathcal{F}$—particularly in some of the elementary formulas which can be included therein—to tighten the bounds on the relations prior to translating the problem for the underlying solver. Any elementary formula that can be represented as a *set containment* (i.e., subset) constraint contains information about the upper and lower bounds of the expressions upon which the constraint is defined. More formally, assume there is an elementary formula defining a subset constraint over expressions ϕ and ρ in the language of $\mathcal{F}$ which is true $\forall m (m \models \mathcal{P})$. Using the definitions in Section 3,

$$\Box\,(\phi \subseteq \rho)$$

Given such a formula, our approach relies on the insights that (a) any element which *must* be assigned to ϕ must also appear in ρ and (b) any element which *could* be assigned to ϕ must also be possible to assign to ρ. That statement is formalized by:

Theorem 1 (Bounding by Subset). *For RL expressions ϕ and ρ,*

$$\Box\,(\phi \subseteq \rho) \implies \left(\lfloor \phi \rfloor \subseteq \lfloor \rho \rfloor \land \lceil \phi \rceil \subseteq \lceil \rho \rceil \right)$$

We prove Theorem 1 by proving two related lemmas, one each for the lower bounds and the upper bounds[1]:

Lemma 1 (Lower Bound Subset)**.** *For expressions ϕ and ρ,*

$$\Box\,(\phi \subseteq \rho) \implies \lfloor \phi \rfloor \subseteq \lfloor \rho \rfloor$$

Proof. The proof starts by assuming that ϕ is necessarily a subset of ρ. Using the rules of modal logic, we can derive the implication that, for some tuple **s**, if it is necessary that **s** is a member of ϕ, it is also necessarily a member of ρ. As that is the definition of a subset relationship, we prove that the lower bound of ϕ must be a subset of the lower bound of ρ. □

Lemma 2 (Upper Bound Subset)**.** *For expressions ϕ and ρ,*

$$\Box\,(\phi \subseteq \rho) \implies \lceil \phi \rceil \subseteq \lceil \rho \rceil$$

Proof. The proof for Lemma 2 is by contradiction, assuming that ϕ is necessarily a subset of ρ, but that the upper bound of ϕ is *not* a subset of the upper bound of ρ. From the latter (and Definition 4), it then follows that there is some tuple, **t**, which could be assigned to ϕ but can never be assigned to ρ. By definition, then there is some model in which that tuple *is* assigned to ϕ but is not assigned to ρ. That model, then, would violate the first half of our original assumption, leading to a contradiction. It follows, then, that the upper bound of ϕ must be a subset of the upper bound of ρ if ϕ is a subset of ρ, proving the lemma. □

Theorem 1 allows us to define rules that tighten the bounds which must also satisfy two additional constraints: (a) lower bounds can only *grow* and (b) upper bounds can only *shrink*. The relational bound propagation approach applies those rules by performing a depth-first, post-order traversal of the abstract syntax tree of the formula $\mathcal{F}$ starting at the root—the complete formula represented by the specification—and terminating when it encounters any elementary formula (i.e., a comparison of one or more expressions)

[1]We include a sketch of each proof here, with a complete Fitch-style proof for Theorem 1 provided as supplementary material on the project website: https://sites.google.com/view/relational-bound-propagation/home [38]

or a negation. For each node in the AST, it performs a different operation depending on the type of node:

- **Propagation:** If the node is a set-containment comparison formula (=, in, or none), the expressions defining the bounds within that subtree can be computed and tightened via bound propagation (described in Section 4.1).
- **Resolution:** If the node is or can be represented as a conjunction (e.g., universal quantification) or a disjunction (e.g., implication), the bound expressions computed for each subtree must be combined either by union (for conjunctions) or intersection (for disjunctions) of the bound expressions from either of the subtrees. Bound resolution is discussed in greater detail in Section 4.2.
- For other operations (e.g., non-set-containment formulas, negations) there is no information to be gained about the bounds, so the traversal is stopped and the provided bound expressions (in $\mathcal{B}$) are used for that subtree.

Algorithm 3 details the overall algorithm used to propagate relational bounds for the top-level formula. When this top-level resolution algorithm encounters a "containment" (e.g., equal, subset) formula, it propagates the bounds for each child expression of the formula then resolves the results of that propagation to compute the new, tightened bounds for the root formula. The tighter bounds can then be used to complete the desired analysis.

4.1 Bound Propagation

The core of the relational bound propagation approach is (a) the propagation of the *relation* bounds in $\mathcal{B}$ to *expressions* in formula $\mathcal{F}$ and (b) the propagation of the tightened bounds for each subtree of any set-containment formulas to the relation bounds. The first step is called *upward propagation* and the second *downward propagation.*

4.1.1 Upward Propagation. Algorithm 1 shows the algorithm used for upward propagation. Upward propagation starts with a depth-first (post-order) traversal of the child expressions of the given expression, $expr$. There are two base cases which stop the traversal: (a) relations (Lines 3-4), which are assumed to be leaves of the tree and return the bounds defined in the passed map (via the Lower and Upper methods which return the lower/upper bounds from the passed bound map, respectively); and (b) expressions which do not match one of the propagation rules and are not propagated. In the latter case (Lines 12-13), the algorithm returns the widest bounds, with an empty lower bound and every possible tuple of the desired arity for the upper bound. For every other case, the upward propagation algorithm computes the *relational expression representing the bounds* for the passed expression based on the bounds of the passed expression's child expressions and according to the *upward propagation rules* defined in Table 1. Rule selection is represented in Algorithm 1 via calls to MatchesUnaryRule or MatchesBinaryRule; these functions select the matching rule from Table 1 based on the operator used in the current expression. Similarly, ApplyUpwardRule finds the applicable rule from Table 1 and applies the rule, computing (and returning) the lower/upper bound for the composite expression. Each rule defines the expression of the bounds for the corresponding operation when applied to child expressions α and (for binary operations) β.

Table 1: Upward propagation rules for relational bound propagation. These sequent-style rules define the bounds for expressions of each relational operator in terms of the bounds of their child expressions (α and/or β). $\downarrow$ or $\uparrow$ denote variables representing a lower or upper bound, respectively.

Operator	Lower Bound	Upper Bound
Transpose	$\dfrac{A_\downarrow = \lfloor \alpha}{\lfloor \alpha^T = (A_\downarrow)^T}$	$\dfrac{A_\uparrow = \lceil \alpha}{\lceil \alpha^T = (A_\uparrow)^T}$
Trans. Closure	$\dfrac{A_\downarrow = \lfloor \alpha}{\lfloor \alpha^+ = (A_\downarrow)^+}$	$\dfrac{A_\uparrow = \lceil \alpha}{\lceil \alpha^+ = (A_\uparrow)^+}$
Reflex. Closure	$\dfrac{A_\downarrow = \lfloor \alpha}{\lfloor \alpha^* = (A_\downarrow)^*}$	$\dfrac{A_\uparrow = \lceil \alpha}{\lceil \alpha^* = (A_\uparrow)^*}$
Union	$\dfrac{A_\downarrow = \lfloor \alpha \qquad B_\downarrow = \lfloor \beta}{\lfloor \alpha \cup \beta = A_\downarrow \cup B_\downarrow}$	$\dfrac{A_\uparrow = \lceil \alpha \qquad B_\uparrow = \lceil \beta}{\lceil \alpha \cup \beta = A_\uparrow \cup B_\uparrow}$
Intersection	$\dfrac{A_\downarrow = \lfloor \alpha \qquad B_\downarrow = \lfloor \beta}{\lfloor \alpha \cap \beta = A_\downarrow \cap B_\downarrow}$	$\dfrac{A_\uparrow = \lceil \alpha \qquad B_\uparrow = \lceil \beta}{\lceil \alpha \cap \beta = A_\uparrow \cap B_\uparrow}$
Difference	$\dfrac{A_\downarrow = \lfloor \alpha \qquad B_\uparrow = \lceil \beta}{\lfloor \alpha \setminus \beta = A_\downarrow \setminus B_\uparrow}$	$\dfrac{A_\uparrow = \lceil \alpha \qquad B_\uparrow = \lceil \beta}{\lceil \alpha \setminus \beta = A_\uparrow \setminus B_\downarrow}$
Product	$\dfrac{A_\downarrow = \lfloor \alpha \qquad B_\downarrow = \lfloor \beta}{\lfloor \alpha \times \beta = A_\downarrow \times B_\downarrow}$	$\dfrac{A_\uparrow = \lceil \alpha \qquad B_\uparrow = \lceil \beta}{\lceil \alpha \times \beta = A_\uparrow \times B_\uparrow}$
Join	$\dfrac{A_\downarrow = \lfloor \alpha \qquad B_\downarrow = \lfloor \beta}{\lfloor \alpha \circ \beta = A_\downarrow \circ B_\downarrow}$	$\dfrac{A_\uparrow = \lceil \alpha \qquad B_\uparrow = \lceil \beta}{\lceil \alpha \circ \beta = A_\uparrow \circ B_\uparrow}$

Most of the rules are straightforward; the bounds for the parent expression can be computed by applying the operation to the corresponding bounds of its children. The justification for each rule follows the same argument. First, by definition, the lower bound of any relation comprises the set of tuples that *must* be assigned to that relation in any satisfying model. It is therefore the *minimal* satisfying assignment for any given relation—if any satisfying model assigned a smaller tuple set, then *that* set would be the lower bound. Hence, for each of these rules, $\lfloor \alpha$ and $\lfloor \beta$ represent the minimal tuple sets to which the operations can be applied—any valid assignment of tuples must be a superset of the lower bounds. The minimal valid assignment for each of the operations can be computed by applying the operation to the minimal valid assignment for each of its operands, so the lower bound for each expression can be computed from the lower bounds of its children. The same argument can be applied in terms of the upper bounds, replacing the smallest or minimal set with the largest or maximal set; the upper bound of each expression can be computed from the upper bounds of its

Algorithm 1 Upward bound propagation algorithm. Bounds for expressions are computed based on the rules defined in Section 4.1.1. Expressions with no matching rule are assigned the widest possible bounds to ensure no solutions are lost.

```
Input: $expr$: a relational expression, $b$: a map of relations in $\mathcal{R}$ to upper and lower bounds, $\mathcal{R}$: a set of relations
Output: $(lower, upper)$: a pair of lower and upper bounds
 1: function UpwardPropagate($expr, b, \mathcal{R}$)
 2:     $t \leftarrow$ typeof $expr$
 3:     if $t$ = relation then
 4:         return $($Lower$(b, expr)$, Upper$(b, expr)$$)$
 5:     else if MatchesUnaryRule($expr$) then
 6:         $(\lfloor x \rfloor, \lceil x \rceil) \leftarrow$ UpwardPropagate(Child($expr$), $b, \mathcal{R}$)
 7:         return ApplyUpwardRule($expr, \lfloor x \rfloor, \lceil x \rceil$)
 8:     else if MatchesBinaryRule($expr$) then
 9:         $(\lfloor l \rfloor, \lceil l \rceil) \leftarrow$ UpwardPropagate(LeftChild($expr$), $b, \mathcal{R}$)
10:         $(\lfloor r \rfloor, \lceil r \rceil) \leftarrow$ UpwardPropagate(RightChild($expr$), $b, \mathcal{R}$)
11:         return ApplyUpwardRule($expr, \lfloor l \rfloor, \lceil l \rceil, \lfloor r \rfloor, \lceil r \rceil$)
12:     else
13:         return $(\emptyset,$ Universe(Arity($expr$))$)$
```

Algorithm 2 Downward bound propagation algorithm. Applies the rules defined in Section 4.1.2 based on expression type.

```
Input: $expr$: a relational expression, $b$: a map of relations in $\mathcal{R}$ to upper and lower bound expressions, $\mathcal{R}$: a set of relations, $lo$: a set of lower bound expressions, $up$: a set of upper bound expressions
 1: function DownwardPropagate($expr, b, \mathcal{R}, lo, up$)
 2:     if $expr$ is relation then
 3:         SetBounds($b, r, lo, up$)
 4:     else if $expr$ is unary then
 5:         $c \leftarrow$ Child($expr$)
 6:         $(lo', up') \leftarrow$ ApplyUnaryRule($expr, lo, up, \lfloor c \rfloor, \lceil c \rceil$)
 7:         DownwardPropagate($c, b, \mathcal{R}, lo', up'$)
 8:     else
 9:         $l \leftarrow$ LeftChild($expr$)
10:         $r \leftarrow$ RightChild($expr$)
11:         $(lo'_l, up'_l) \leftarrow$ ApplyLeftRule($expr, lo, up, \lfloor l \rfloor, \lceil l \rceil, \lfloor r \rfloor, \lceil r \rceil$)
12:         $(lo'_r, up'_r) \leftarrow$ ApplyRightRule($expr, lo, up, \lfloor l \rfloor, \lceil l \rceil, \lfloor r \rfloor, \lceil r \rceil$)
13:         DownwardPropagate($l, b, \mathcal{R}, lo'_l, up'_l$)
14:         DownwardPropagate($r, b, \mathcal{R}, lo'_r, up'_r$)
```

children. The sole exception is the difference operator. In that case, the minimal set for the expression can be computed by subtracting the *maximal* set for the right child from the *minimal* set for the left child, α. The maximal set can be computed by subtracting the minimal set for the right from the maximal set for the left.

By applying these rules as described in Algorithm 1, the relational bounds propagate up the AST of the formula from the leaves (i.e., the relations) to the set containment formula. The bounds for each expression are then passed back *down* the tree.

4.1.2 Downward Propagation. Having determined the expressions for the lower and upper bounds for each child expression of a set-containment formula, the bounds can be propagated back down the subtree to tighten the bounds on each relation; only the *relation* bounds are passed to the solver. Algorithm 2 details the process followed for downward propagation of bound expressions, again by performing a depth-first traversal of the AST for each relational expression. For each expression $x \equiv \alpha \, \mathbf{op} \, \beta$, the bound expressions are propagated from x to both α and β according to the *downward propagation rules* defined in Table 2; this action is represented in Algorithm 2 by invocations of the Apply*Rule methods (lines 6, 11, and 12). If the expression is a relation, the bounds for that relation are updated in the passed bound map (Line 3). For unary (e.g., transpose) or commutative (e.g., union, intersection) operations, the rule is presented for one child (i.e., α). As each rule must satisfy the conditions that lower bounds only grow and upper bounds only shrink, the the rules assign the result of a set union and a set intersection/difference to the lower/upper bound, respectively [2].

- **Transpose**: The transpose operation is applied to the bounds of the parent and the result is propagated down to the corresponding bound for the child.
- **Closure**: For transitive and reflexive closure, no information can be gained about the lower bound, and the upper bound of the child is constrained by that of the parent.
- **Union**: For union, any tuple in the lower bound of the union that *cannot* appear in one child can be safely added to the lower bound of the other (as it must appear there in order to guarantee its presence in the parent). Any tuple that does not appear in the upper bound of the parent can be removed from the upper bounds of either child.
- **Intersection**: The intersection contains all tuples in *both* children, so the tuples in the lower bound of the intersection can be added to the lower bounds of each child. Further, any tuple that appears in the *lower* bound of one child but does *not* appear in the upper bound of the parent can be safely removed from the upper bound of the other child.
- **Difference**: For set difference, the expression contains any tuple from the left child that does not appear in the right child. Thus, any tuple that is in the lower bound of the difference expression can be added to the lower bound of the left child. Furthermore, any child that does not appear in the upper bound of the parent *and* does not appear in the upper bound of the right child can be removed from the upper bound of the left child. For the right child, any child that must appear in the left child but must *not* appear in the parent can be added to the lower bound, as it must appear in the right child in order to remove it from the result. Similarly, any tuple that *must* appear in the result can be removed from the upper bound of the right child.

The downward propagation rules for the product of two relations rely on some additional transformations to extract the prefix and suffix from the tuples in the expression's bounds.

Definition 5 (*n*-prefix Relation)**.** *The n-prefix relation of a given relation ϕ of arity m greater than or equal to some positive integer n—denoted $PR(n, \phi)$—comprises the set of all n-tuples defining a prefix of a tuple in ϕ. Formally, the n-prefix relation is defined as follows:*

$$PR(n, \phi) \equiv \{ (x_1, ..., x_n) \mid \exists \mathrm{y} \, (\mathrm{y} \in \phi \land \bigwedge_{i=1}^{n} y_i = x_i) \}$$

[2]The proofs for each downward propagation rule are included in the supplementary material available on the project website: https://sites.google.com/view/relational-bound-propagation/home [38].

Table 2: Downward propagation rules for relational bound propagation. These sequent style rules adjust the bounds for children of relational expressions using each operator (for expression γ, γ' indicates the adjusted value after applying bound propagation). For unary/commutative operators, only one rule is provided, covering both upper/lower bound. For others, one rule is for the left child and one for the right. Helper predicates (e.g., PR) are defined in Section 4. ↓ / ↑ denote lower/upper bound, respectively.

Operator	Lower Bound	Upper Bound
Transpose	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad R_\downarrow = \lfloor \alpha^T \rfloor}{\lfloor \alpha \rfloor' = A_\downarrow \cup (R_\downarrow)^T}$	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad R_\uparrow = \lceil \alpha^T \rceil}{\lceil \alpha \rceil' = A_\uparrow \cap (R_\uparrow)^T}$
Trans. Closure	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad R_\downarrow = \lfloor \alpha^+ \rfloor}{\lfloor \alpha \rfloor' = A_\downarrow}$	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad R_\uparrow = \lceil \alpha^+ \rceil}{\lceil \alpha \rceil' = A_\uparrow \cap R_\uparrow}$
Reflex. Closure	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad R_\downarrow = \lfloor \alpha^* \rfloor}{\lfloor \alpha \rfloor' = A_\downarrow}$	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad R_\uparrow = \lceil \alpha^* \rceil}{\lceil \alpha \rceil' = A_\uparrow \cap R_\uparrow}$
Union	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad B_\uparrow = \lceil \beta \rceil \quad R_\downarrow = \lfloor \alpha \cup \beta \rfloor}{\lfloor \alpha \rfloor' = A_\downarrow \cup (R_\downarrow \setminus B_\uparrow)}$	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad R_\uparrow = \lceil \alpha \cup \beta \rceil}{\lceil \alpha \rceil' = A_\uparrow \cap R_\uparrow}$
Intersection	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad R_\downarrow = \lfloor \alpha \cap \beta \rfloor}{\lfloor \alpha \rfloor' = A_\downarrow \cup R_\downarrow}$	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad B_\downarrow = \lfloor \beta \rfloor \quad R_\uparrow = \lceil \alpha \cap \beta \rceil}{\lceil \alpha \rceil' = A_\uparrow \cap \left(R_\uparrow \cup (B_\downarrow)^\complement\right)}$
Difference (Left)	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad R_\downarrow = \lfloor \alpha \setminus \beta \rfloor}{\lfloor \alpha \rfloor' = A_\downarrow \cup R_\downarrow}$	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad B_\uparrow = \lceil \beta \rceil \quad R_\uparrow = \lceil \alpha \setminus \beta \rceil}{\lceil \alpha \rceil' = A_\uparrow \cap (R_\uparrow \cup B_\uparrow)}$
Difference (Right)	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad B_\downarrow = \lfloor \beta \rfloor \quad R_\uparrow = \lceil \alpha \setminus \beta \rceil}{\lfloor \beta \rfloor' = B_\downarrow \cup (A_\downarrow \setminus R_\uparrow)}$	$\dfrac{B_\uparrow = \lceil \beta \rceil \quad R_\downarrow = \lfloor \alpha \setminus \beta \rfloor}{\lceil \beta \rceil' = B_\uparrow \cap (R_\downarrow)^\complement}$
Product (left)	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad R_\downarrow = \lfloor \alpha \times \beta \rfloor}{\lfloor \alpha \rfloor' = A_\downarrow \cup \mathrm{PR}(\textsc{arity}(\alpha), R_\downarrow)}$	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad R_\uparrow = \lceil \alpha \times \beta \rceil \quad \lfloor \beta \rfloor \neq \emptyset}{\lceil \alpha \rceil' = A_\uparrow \cap \mathrm{PR}(\textsc{arity}(\alpha), R_\uparrow)}$
Product (right)	$\dfrac{B_\downarrow = \lfloor \beta \rfloor \quad R_\downarrow = \lfloor \alpha \times \beta \rfloor}{\lfloor \beta \rfloor' = B_\downarrow \cup \mathrm{SR}(\textsc{arity}(\beta), R_\downarrow)}$	$\dfrac{B_\uparrow = \lceil \beta \rceil \quad R_\uparrow = \lceil \alpha \times \beta \rceil \quad \lfloor \alpha \rfloor \neq \emptyset}{\lceil \beta \rceil' = B_\uparrow \cap \mathrm{SR}(\textsc{arity}(\beta), R_\uparrow)}$
Join (left)	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad A_\uparrow = \lceil \alpha \rceil \quad B_\downarrow = \lceil \beta \rceil \quad R_\downarrow = \lfloor \alpha \circ \beta \rfloor}{\lfloor \alpha \rfloor' = A_\downarrow \cup \{\, \mathbf{a} \mid \exists \mathbf{r}\, [\, \mathbf{r} \in R_\downarrow \wedge \mathrm{OP}(\mathbf{a}, A_\uparrow, B_\uparrow, \mathbf{r})\,]\,\}}$	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad B_\downarrow = \lfloor \beta \rfloor \quad R_\uparrow = \lceil \alpha \circ \beta \rceil}{\lceil \alpha \rceil' = A_\uparrow \setminus \{\, \mathbf{a} \mid \mathbf{a} \in A_\uparrow \wedge \exists \mathbf{r}\, [\, \mathrm{JP}(\mathbf{a}, B_\downarrow, \mathbf{r}) \wedge \mathbf{r} \notin R_\uparrow\,]\,\}}$
Join (right)	$\dfrac{A_\uparrow = \lceil \alpha \rceil \quad B_\downarrow = \lfloor \beta \rfloor \quad B_\uparrow = \lceil \beta \rceil \quad R_\downarrow = \lfloor \alpha \circ \beta \rfloor}{\lfloor \beta \rfloor' = B_\downarrow \cup \{\, \mathbf{b} \mid \exists \mathbf{r}\, [\, \mathbf{r} \in R_\downarrow \wedge \mathrm{OS}(\mathbf{b}, A_\uparrow, B_\uparrow, \mathbf{r})\,]\,\}}$	$\dfrac{A_\downarrow = \lfloor \alpha \rfloor \quad B_\uparrow = \lceil \beta \rceil \quad R_\uparrow = \lceil \alpha \circ \beta \rceil}{\lceil \beta \rceil' = B_\uparrow \setminus \{\, \mathbf{b} \mid \mathbf{b} \in B_\uparrow \wedge \exists \mathbf{r}\, [\, \mathrm{JS}(\mathbf{b}, A_\downarrow, \mathbf{r}) \wedge \mathbf{r} \notin R_\uparrow\,]\,\}}$

Definition 6 (n-suffix Relation). *The n-suffix relation of a given relation ϕ of arity m greater than or equal to some positive integer n—denoted $SR(n, \phi)$—comprises the set of all n-tuples defining a suffix of a tuple in ϕ. Formally, the n-suffix relation is defined as follows:*

$$SR(n, \phi) \equiv \{(x_1, ..., x_n) \mid \exists \mathbf{y}(\mathbf{y} \in \phi \wedge \bigwedge_{i=0}^{n-1} y_{m-i} = x_{n-i}\}$$

Similarly, the downward propagation rules for the join operator rely on additional predicates to determine whether a given tuple from the upper bound of α should be included in its lower bound based on the tuples in the lower bound of the join expression. In this case, the lower bound of α (the left child of the expression) expands to include any tuple from the upper bound of α which is the *only* tuple that would be a valid prefix for a tuple appearing the lower bound of the join expression, assuming it is joined to the upper bound of β (similarly for β and suffixes). The definition of a “prefix” and a “suffix” is slightly different for the relational join as well, as the join operation drops both the last atom from the tuples in the left expression and the first atom from the tuples in the right expression. Those additional prefix/suffix predicates are defined as:

Definition 7 (Join Prefix)**.** *The predicate* join-prefix *determines whether a given tuple* $\mathbf{a}$ *of arity* n *can be joined with any tuple from relation* β *with arity* m *such that the result is equal to the tuple given by* $\mathbf{r}$ *(of arity* $m + n - 2$*). Formally:*

$$\mathcal{JP}(\mathbf{a}, \beta, \mathbf{r}) \equiv \bigwedge_{i=1}^{n-1} (r_i = a_i) \wedge \exists \mathbf{b}(\mathbf{b} \in \beta \wedge \bigwedge_{j=2}^{m} (r_{n+j-2} = b_j))$$

Definition 8 (Only Prefix)**.** *The predicate* only-prefix *determines whether a given tuple* $\mathbf{a}$ *drawn from relation* α *of arity* n *is the* only *tuple in* α *that can be joined with a tuple from relation* β *of arity* m *such that the result is equal to the tuple given by* $\mathbf{r}$. *Formally:*

$$\begin{aligned} OP(\mathbf{a}, \alpha, \beta, \mathbf{r}) \equiv\ & \mathbf{a} \in \alpha \wedge \mathcal{JP}(\mathbf{a}, \beta, \mathbf{r}) \\ & \wedge \neg\exists \mathbf{x}\,(\mathbf{x} \in \alpha \wedge \mathbf{x} \neq \mathbf{a} \wedge \mathcal{JP}(\mathbf{x}, \beta, \mathbf{r})\,) \end{aligned}$$

Definition 9 (Join Suffix)**.** *The predicate* join-suffix *determines whether a given tuple* $\mathbf{b}$ *of arity* m *can be joined (as the suffix) to any tuple from relation* α *with arity* n *such that the result is equal to the tuple given by* $\mathbf{r}$ *(of arity* $m + n - 2$*). Formally:*

$$\mathcal{JS}(\mathbf{b}, \alpha, \mathbf{r}) \equiv \bigwedge_{j=2}^{m} (r_{j+n-2} = b_j) \wedge \exists \mathbf{a}(\mathbf{a} \in \alpha \wedge \bigwedge_{i=1}^{n-1} (r_i = a_i))$$

Definition 10 (Only Suffix)**.** *The predicate* only-suffix *determines whether a given tuple* $\mathbf{b}$ *drawn from relation* β *of arity* m *is the* only *tuple in* β *that can be joined (as the suffix) to a tuple from relation* α *of arity* n *such that the result is equal to the tuple given by* $\mathbf{r}$. *Formally:*

$$\begin{aligned} OS(\mathbf{b}, \alpha, \beta, \mathbf{r}) \equiv\ & \mathbf{b} \in \beta \wedge \mathcal{JS}(\mathbf{b}, \alpha, \mathbf{r}) \\ & \wedge \neg\exists \mathbf{y}\,(\mathbf{y} \in \beta \wedge \mathbf{y} \neq \mathbf{b} \wedge \mathcal{JS}(\mathbf{y}, \alpha, \mathbf{r})\,) \end{aligned}$$

4.2 Bound Resolution

After applying the downward propagation rules to adjust the bounds of the relations in each subtree, the bounds for each subtree must be merged with the bounds propagated through sibling subtrees via bound resolution. Algorithm 3 describes the resolution process, which applies a simple rule to combine the bounds from each subtree based on whether the sibling subtrees are children of a conjunctive or disjunctive formula. For the purposes of our algorithm, *conjunctive* formulas can be equivalently expressed as a conjunction (e.g., biconditionals, universal quantifiers); *disjunctive* formulas can be expressed as a disjunction (e.g., if); and *containment* formulas represent a set containment comparison (e.g., subset, equals). In the case of a conjunctive formula (lines 6-12), the bounds can be tightened by taking the union of the lower bound for each relation and the intersection of the upper bound for each relation. As each child statement must be true, the bounds computed for each must hold in the other. For disjunctive formulas (lines 13-19), the bounds are "loosened", taking the most permissive bound from the two subtrees. This is done by taking the intersection of the lower bounds and the union of the upper bounds, such that only those bounds that apply in both subtrees are preserved. The resolved bounds are then passed back up the AST to the ancestors of the resolved subtrees for further resolution, all the way to the root. Once the algorithm reaches the root, the bounds that have been thus computed are used as the bounds for the original relational bounded model checking problem.

Algorithm 3 Bound resolution algorithm. Performs a recursive depth-first (post-order) traversal of the AST of f and resolves the upper and lower bounds for every relation in $\mathcal{R}$.

Input: f: relational formula, b: map of relations in $\mathcal{R}$ to upper and lower bound expressions, $\mathcal{R}$: set of relations
Output: b': map of relations to adjusted upper/lower bound expressions

```
1:  function Resolve($f, b, \mathcal{R}$)
2:      $b' \leftarrow b$
3:      $l \leftarrow$ LeftChild($f$)
4:      $r \leftarrow$ RightChild($f$)
5:      switch typeof $f$ do
6:          case conjunctive
7:              $b_1 \leftarrow$ Resolve($l, b', \mathcal{R}$)
8:              $b_2 \leftarrow$ Resolve($r, b', \mathcal{R}$)
9:              for $r \in \mathcal{R}$ do
10:                 $lo \leftarrow$ Lower($b_1, r$) $\cup$ Lower($b_2, r$)
11:                 $up \leftarrow$ Upper($b_1, r$) $\cap$ Upper($b_2, r$)
12:                 SetBounds($b', r, lo, up$)
13:         case disjunctive
14:             $b_1 \leftarrow$ Resolve($l, b', \mathcal{R}$)
15:             $b_2 \leftarrow$ Resolve($r, b', \mathcal{R}$)
16:             for $r \in \mathcal{R}$ do
17:                 $lo \leftarrow$ Lower($b_1, r$) $\cap$ Lower($b_2, r$)
18:                 $up \leftarrow$ Upper($b_1, r$) $\cup$ Upper($b_2, r$)
19:                 SetBounds($b', r, lo, up$)
20:         case containment
21:             $(\lfloor l \rfloor, \lceil l \rceil) \leftarrow$ UpwardPropagate($l, b', \mathcal{R}$)
22:             $(\lfloor r \rfloor, \lceil r \rceil) \leftarrow$ UpwardPropagate($r, b', \mathcal{R}$)
23:             $b' \leftarrow$ DownwardPropagate($l, b', \mathcal{R}, \lfloor l \rfloor, \lceil l \rceil \cap \lceil r \rceil$)
24:             $b' \leftarrow$ DownwardPropagate($r, b', \mathcal{R}, \lfloor l \rfloor \cup \lfloor r \rfloor, \lceil r \rceil$)
25:         else
26:             break
27:     return $b'$
```

5 EVALUATION

The experimental evaluation conducted for relational bound propagation seeks the answers to three research questions using Propter, our realization of relational bound propagation:

RQ1. Does Propter reduce the problem size for relational bounded model checking problems?
RQ2. Does Propter reduce analysis time for large-scale real-world relational bounded model checking problems?
RQ3. How much overhead does relational bound propagation add?

Experimental setup. The experiments were conducted using Propter, a custom Java 19 tool implementing relational bound propagation as described in Section 4 and comprising over 4,500 lines of code. Propter is available on the project's website for free use in the academic community [38]. The specifications used in the experiments were developed in Alloy [20] and executed using the Java API of Alloy 5.1, the Kodkod model finder [40] which drives that version of the Alloy Analyzer, and the MiniSAT [34] SAT solver. All experiments were run on an OpenStack Ubuntu 20.04 instance

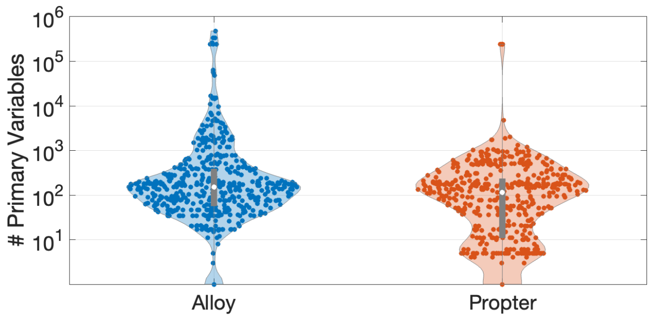


Figure 4: # primary variables (log scale). Propter reduced primary variables by an average of 30.68% (N=519).

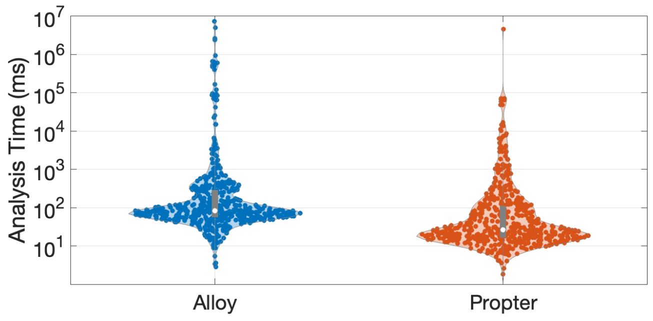


Figure 5: Translate + solve time (log ms). Propter reduced translate + solve time by 49.30% on average (N=519).

with 16 2.3GHz VCPUs and 100GB RAM inside a Docker 20.10.17 container extending Eclipse Temurin Java 19.

Subject systems. As subjects, our experiments were run on a collection of 401 Alloy specifications drawn or adapted from prior studies [1, 5, 8, 19, 21, 23, 29] covering a variety of real-world problem domains: IoT and Android app security analysis [1, 5, 8]; database design [29]; and role-based access control policy verification [19, 21]. We also included a set of example Alloy specifications curated by the Alloy Tools team [23]. These specifications included 519 analyzable commands (i.e., formal analysis problems), 401 satisfiable and 118 unsatisfiable. Each command was analyzed five times with the default bounds generated by Alloy 5.1 and five times with bounds tightened using relational bound propagation.

Baselines and measures. Our experiments measured five properties related to our three research questions. First, we measured (a) the number of primary variables and (b) the total number of variables in the CNF problem presented to the underlying SAT solver when performing the analysis for each specification to answer RQ1. We also tracked the time required to (c) translate each Alloy specification into a CNF problem and (d) perform the analysis with the SAT solver (both in milliseconds). Finally, we recorded (e) the time taken (in milliseconds) to propagate the bound expressions for each specification as a measure of the overhead (for RQ3). We compare Propter (our custom implementation of relational bound propagation) against an unmodified distribution of Alloy 5.1 using the MiniSAT SAT solver [34].

5.1 RQ1: Reduction in Problem Size

To answer RQ1, we examined the reduction in the scope of each bounded relational model checking problem in terms of the number of *primary variables* in the relational bounded model checking problem. These variables provide an indication of the total size of the problem; each variable represents a tuple assignment to a specific relation that is either present or not present in given model of the specification. We recorded the number of primary variables both before and after relational bound propagation. Figure 4 summarizes the number of primary variables in each subject before and after relational bound propagation. The median number of primary variables before and after were 151 and 114, respectively. Propter reduced the number of primary variables in 330 of the 519 subjects (63.58%), 164 of them (31.60%) by 50% or more. The mean reduction in the number of primary variables was 30.68%. For 19 of the commands (3.80%), Propter was able to provide *tight bounds* for the problem, solving the problem without using the SAT solver; the largest such case was assigned 478,400 primary variables by default, reduced to 0 by Propter. To demonstrate the practical impact of relational bound propagation, we also measured the time taken to translate and solve each formal analysis problem with Alloy and with Propter. The reduction in the primary variables reduced the size of each problem, so we would expect to see a similar reduction in the analysis time due to the reduced problem size. Figure 5 summarizes the translation and solving time for each of the 519 commands before and after performing bound propagation with Propter, excluding overhead. Propter was able to reduce the translation and solve time for 466 of the subject commands (89.79%), with a mean reduction of 49.30%. These results demonstrate that **relational bound propagation effectively reduces the size of formal analysis problems for bounded relational logic specifications.**

5.2 RQ2: Large-scale Real-world Specifications

To answer RQ2, we conducted a more detailed analysis of the timing results for large-scale specifications resulting in CNF representations with more than 100,000 variables (25 of 519 commands). These 25 commands came from the specifications drawn from real-world systems [1, 5, 8, 19, 33]. The results of applying Propter to these 25 commands are summarized in Figure 6. The figure shows box plots summarizing the total time—including both the analysis and overhead—required to execute each command, grouped by the base-10 log of the total number of variables in each command's CNF translation. The group labeled 10^5, for example, contains all commands with more than 100,000 variables but fewer than one million, and so on. Propter reduced the total analysis time for each group taken individually and across all 25 commands, Propter reduced the total analysis time by an average of 68.14%, from (on average) 775.355 seconds with Alloy Analyzer to 226.543 seconds with Propter. For the largest specifications (with more than 100 million CNF variables) Propter reduced the average total analysis time by 99.51%, from just over 2 hours (7,378.4s) to 36.4s. In total, across all 125 analyses of large-scale, real-world specifications (5 each for each of the 25 commands), Propter saved 65,627.117 seconds (18.23

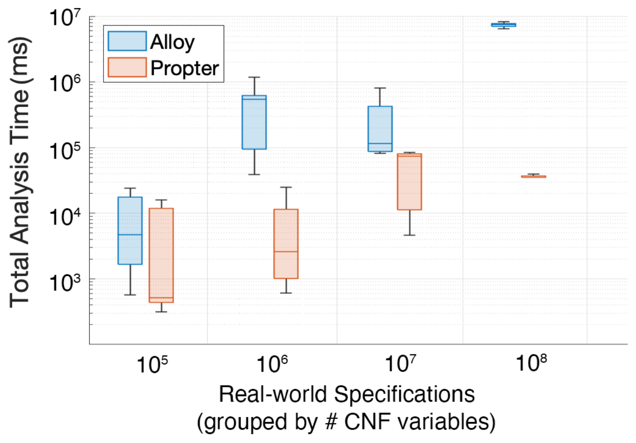


Figure 6: Total analysis times (in log ms, including overhead) for 25 large-scale, real-world analysis commands, grouped by log total # variables in each command's CNF translation. Each command was analyzed 5 times with Alloy (left box, blue) and 5 times with Propter (right box, orange). Propter outperforms Alloy in each group, most significantly so for the largest commands (with $> 10^8$ variables).

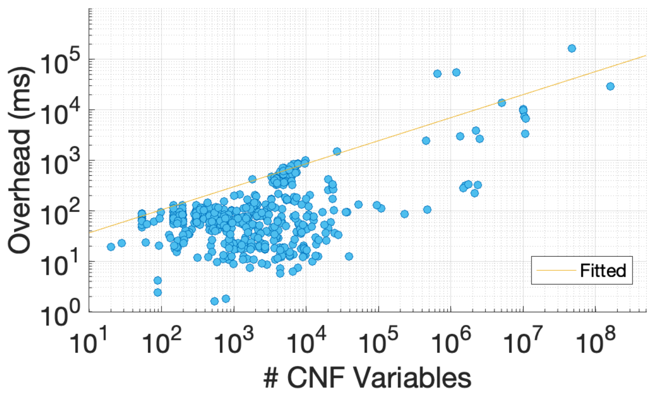


Figure 7: Overhead of relational bound propagation (in ms, log scale) vs. total # variables (log scale) in the resulting CNF for 519 analysis commands. Solid line indicates the power curve fitted to the scatter data ($y = 12.79x^{0.4565}$).

hours) of total analysis time compared to Alloy Analyzer. We interpret these results to show that **relational bound propagation can effectively reduce the time required to perform formal analysis of real-world system specifications such as those used in our experiments.**

5.3 RQ3: Overhead

Relational bound propagation introduces additional computational overhead compared to the baseline. We computed the overhead by calculating the total time taken to analyze each subject formal analysis command with Propter and subtracting the time taken to (a) translate the command into CNF and (b) to solve the command with the underlying SAT solver. The remaining time is considered to be the overhead introduced by relational bound propagation. Figure 7 shows the overhead (in log milliseconds) compared to the total number of variables in the baseline translation to CNF (i.e., without relational bound propagation). Overall, Propter introduces very little overhead, with the vast majority of the commands (499 of 519, or 96.15%) requiring less than one second to propagate the bounds. The solid line on the graph shows the fitted power series describing the growth of the overhead, $y = 12.79x^{0.4565}$. For the 25 large-scale, real-world specifications used to answer RQ2, Propter added little overhead compared to its benefits. For one command, relational bound propagation took a significant amount of time with respect to the baseline analysis time (51.745s of overhead vs. 22.420s of baseline analysis time). For the other 24, overhead was on average 6.14% of the baseline analysis time (14.318s), vs. an average time savings of 68.14% (54.881s). We interpret these results to show that, **for large-scale specifications, the overhead required for relational bound propagation is small compared to the savings in analysis time.**

6 DISCUSSION

The experiments in Section 5 demonstrate Propter improves the analysis of large-scale, real-world specifications. First, relational bound propagation exponentially reduces the size of the search space by reducing the number of primary variables in each problem. Each primary variable represents a Boolean value passed to the underlying solver; removing even one primary variable cuts the size of the search space in half. This can be very important for the real-world specifications such as those we used for our evaluation. These are often generated from source code (e.g., Android [5], IoT [1]) or created by non-experts, resulting in specifications (and bounds) that are poorly optimized. *Propter reduced the number of primary variables by an average of 30.68%*, which leads to a huge reduction in the search space for the solver. This leads to a corresponding reduction in the analysis time, which is especially impactful for specifications based on real-world systems. For many of the IoT and RBAC specifications, for example, Propter reduced the average total analysis time by *two orders of magnitude*, saving nearly two hours of analysis time for our largest subjects. These results demonstrate Propter can improve the scalability of relational bounded model checking for real-world software systems.

Limitations: The rules as defined in Section 4 provide translations for the standard n-ary relational operators, but those rules do not tighten the bounds for two constructs available to relational specifications—*comprehension* expressions and *quantified* expressions. For comprehensions and quantifications, Propter uses the default bounds in place of the computed bounds for each expression variable (the empty set for the lower bound and the full Cartesian product of the universe of atoms of the desired arity for the upper bound). This preserves the correctness of the relational bound propagation, but a full treatment of comprehensions and quantifications could benefit more complex specifications. Similarly, the bound resolution algorithm is limited to subformulas not contained within a negation; such subtrees are assumed to use the default, unmodified bounds and are not traversed during resolution. It may be possible to resolve the bounds even within those negated subtrees, possibly by performing some additional transformations to push the negations lower in the tree.

Threats to Validity: The internal validity of these experiments relies on the accuracy of our Java implementation as well as our choice of other tools to compare against. To reduce the threat posed by our implementation we optimized Propter to ensure the validity of our conclusions and tested it rigorously. Moreover, each of the propagation rules is provably valid, which ensures that the tightened bounds are correct. For our comparison, we chose to compare Propter against the analyzer included with Alloy 5.1. We believe this serves as a valid comparison because (a) other tools which optimize via bound tightening require either a specific domain or additional inputs (as described in Section 7) and (b) *no* other tool, to the best of our knowledge, employs any technique which is theoretically similar to relational bound propagation. Therefore, a comparison to Alloy 5.1 is the best way to test for improvements attributable to relational bound propagation alone. To mitigate any threats to our external validity, we have studied hundreds of Alloy specifications derived from various sources, the majority of which relate to real-world software systems. While we cannot claim they are representative of all such specifications, the variety of problem domains (e.g., Android app security analysis, role-based access control) and specification sources (e.g., the specifications from [1] were automatically extracted from real IoT apps) help to determine whether our results may generalize. Lastly, the measures used in the experiments directly correspond to the qualities being evaluated by the research questions, ensuring the validity of our construct.

7 RELATED WORK

Researchers have recently proposed a variety of techniques that also employ bound tightening, albeit for a variety of purposes [4, 10, 16, 26]. Bagheri and Malek [4] introduced Titanium, which (as part of its compositional analysis) enumerates all models for a specification and stores the “observed” bounds for later use. A similar technique is used by Flair [10, 11], which also uses the observed bounds to reduce the search space used in its analysis. Propter takes a different approach; rather than *enumerating the entire model space* in order to tighten the bounds—which can be very expensive—Propter applies the rules described above to tighten the bounds *a priori*, without the cost of having run the analysis first. Galeotti, et al. [16] use symmetry breaking to determine tight bounds for TACO, their tool to analyze annotated Java code. Stevens and Bagheri [37] presented an approach that is more general, but requires the specification of additional elements (namely the domain-specific operations) and is targeted to repeated analysis of changing specifications. Relational bound propagation has no such limitation, as it can be applied to any specification without modification or additional specification and does not assume any particular use case surrounding the analysis. As these other tools are limited in domain or require additional specification, we did not include them in our experiments.

Few researchers have approached the problem by examining the relational formula extracted from the specification. A recent study by Wang, et al. [42] uses machine learning to choose an optimal SAT solver for a given specification, but does not address the bounds themselves. In another study, Wang et al. [41] present some rules to simplify relational formulas by replacing expressions within a formula with semantically equivalent expressions. In that study, they cite the only simplification rule currently implemented in the Alloy Analyzer—namely that $a\,.\,(a \times b)$ is semantically equivalent to b when the cardinality of a is greater than zero. While their work explores the semantics of relational expressions, they evaluate semantic equivalence in order to generate formulas that are *non*-equivalent for the purposes of formula synthesis. We instead seek to use these transformations to tighten the bounds for analysis. To the best of our knowledge, this paper is the first work to explore applying transformation rules and propagation techniques in the context of relational bounds. Importantly, the prior tools mentioned here were not developed for the purpose of improving the underlying formal analysis and as such were not included as comparisons in our experiments.

8 CONCLUSION

This paper introduces *relational bound propagation*, a novel method to automatically—and validly—tighten the bounds used for relational bounded model checking by extracting information from the relational formula in the problem itself. Our approach applies a set of upward- and downward-propagation rules to tighten the default bounds and reduce the size of the search space. We performed an experimental evaluation of an implementation of those transformation rules on real-world Alloy specifications drawn from other sources in the literature. Our results show that relational bound propagation improves the scalability of formal analysis on real-world specifications vs. the baseline, in many cases reducing the problem to one with tight bounds (30.68% on average). Relational bound propagation reduced the analysis time for real-world specifications by an average of 68.14%, all with very low overhead. In future research, we plan to improve upon the current approach by improving and extending the propagation rules described here. We will seek to find ways to combine relational bound propagation with other techniques which manipulate relational bounds (e.g., [37]). Finally, this work relies heavily on deterministic logics; we believe there may be opportunities to explore other systems such as fuzzy logics [24, 30].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work was completed utilizing the Holland Computing Center of the University of Nebraska, which receives support from the Nebraska Research Initiative. This work was supported in part by awards CCF-1755890, CCF-1618132, CCF-2139845, and CCF-2124116 from the National Science Foundation.

REFERENCES

[1] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. 2020. Scalable analysis of interaction threats in IoT systems. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 272–285. https://doi.org/10.1145/3395363.3397347

[2] Mohannad Alhanahnah, Clay Stevens, Bocheng Chen, Qiben Yan, and Hamid Bagheri. 2023. IoTCom: Dissecting Interaction Threats in IoT Systems. *IEEE Trans. Software Eng.* 49, 4 (2023), 1523–1539. https://doi.org/10.1109/TSE.2022.3179294

[3] Hamid Bagheri, Eunsuk Kang, and Niloofar Mansoor. 2020. Synthesis of assurance cases for software certification. In *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 61–64. https://doi.org/10.1145/3377816.3381728

[4] Hamid Bagheri and Sam Malek. 2016. Titanium: efficient analysis of evolving alloy specifications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 27–38. https://doi.org/10.1145/2950290.2950337

[5] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Trans. Software Eng.* 41, 9 (2015), 866–886. https://doi.org/10.1109/TSE.2015.2419611

[6] Hamid Bagheri and Kevin J. Sullivan. 2012. Pol: specification-driven synthesis of architectural code frameworks for platform-based applications. In *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, Klaus Ostermann and Walter Binder (Eds.). ACM, 93–102. https://doi.org/10.1145/2371401.2371416

[7] Hamid Bagheri and Kevin J. Sullivan. 2016. Model-driven synthesis of formally precise, stylized software architectures. *Formal Aspects Comput.* 28, 3 (2016), 441–467. https://doi.org/10.1007/S00165-016-0360-8

[8] Hamid Bagheri, Chong Tang, and Kevin J. Sullivan. 2014. TradeMaker: automated dynamic analysis of synthesized tradespaces. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 106–116. https://doi.org/10.1145/2568225.2568291

[9] Hamid Bagheri, Chong Tang, and Kevin J. Sullivan. 2017. Automated Synthesis and Dynamic Analysis of Tradeoff Spaces for Object-Relational Mapping. *IEEE Trans. Software Eng.* 43, 2 (2017), 145–163. https://doi.org/10.1109/TSE.2016.2587646

[10] Hamid Bagheri, Jianghao Wang, Jarod Aerts, Negar Ghorbani, and Sam Malek. 2021. Flair: efficient analysis of Android inter-component vulnerabilities in response to incremental changes. *Empir. Softw. Eng.* 26, 3 (2021), 54. https://doi.org/10.1007/s10664-020-09932-6

[11] Hamid Bagheri, Jianghao Wang, Jarod Aerts, and Sam Malek. 2018. Efficient, Evolutionary Security Analysis of Interacting Android Apps. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 357–368. https://doi.org/10.1109/ICSME.2018.00044

[12] Patrick Blackburn, Johan van Benthem, and Frank Wolter. 2006. *Handbook of Modal Logic.* Elsevier.

[13] Antonio Bucchiarone and Juan P. Galeotti. 2008. Dynamic Software Architectures Verification using DynAlloy. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 10 (2008). https://doi.org/10.14279/tuj.eceasst.10.145

[14] Javier Cámara, David Garlan, and Bradley R. Schmerl. 2019. Synthesizing tradeoff spaces with quantitative guarantees for families of software systems. *Journal of Systems and Software* 152 (2019), 33–49. https://doi.org/10.1016/j.jss.2019.02.055

[15] Byron Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[16] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE Trans. Software Eng.* 39, 9 (2013), 1283–1307. https://doi.org/10.1109/TSE.2013.15

[17] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. 2009. Distributed SAT-Based Computation of Relational Tight Bounds. In *APV 2009: Intl. Symposium on Automatic Program Verification, Río Cuarto, Argentina, Feb 15, 2009, Unpublished.* http://se.inf.ethz.ch/old/events/apv/files/APV09-02.pdf

[18] Klaus Marius Hansen and Mads Ingstrup. 2010. Modeling and analyzing architectural change with alloy. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung (Eds.). ACM, 2257–2264. https://doi.org/10.1145/1774088.1774560

[19] Ramzi A. Haraty and Mirna Naous. 2013. Role-Based Access Control modeling and validation. In *2013 IEEE Symposium on Computers and Communications, ISCC 2013, Split, Croatia, 7-10 July, 2013*. IEEE Computer Society, 61–66. https://doi.org/10.1109/ISCC.2013.6754925

[20] D. Jackson. 2012. *Software Abstractions* (2nd ed.). MIT Press.

[21] Somesh Jha, Ninghui Li, Mahesh V. Tripunitara, Qihua Wang, and William H. Winsborough. 2008. Towards Formal Verification of Role-Based Access Control Policies. *IEEE Trans. Dependable Secur. Comput.* 5, 4 (2008), 242–255. https://doi.org/10.1109/TDSC.2007.70225

[22] Jung Soo Kim and David Garlan. 2006. Analyzing architectural styles with alloy. In *Proceedings of the 2006 Workshop on Role of Software Architecture for Testing and Analysis, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), ROSATEA 2006, Portland, Maine, USA, July 17-20, 2006*, Robert M. Hierons and Henry Muccini (Eds.). ACM, 70–80. https://doi.org/10.1145/1147249.1147259

[23] Peter Kriens, Sergey Bronnikov, Burkhardt Renz, and Daniel Jackson. 2023. Alloy models GitHub repository. https://github.com/AlloyTools/models.

[24] Dong Liu and Benjamin Carrión Schäfer. 2016. Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs. In *Proceedings of FPL*. 1–8.

[25] Niloofar Mansoor, Jonathan A. Saddler, Bruno Vieira Resende e Silva, Hamid Bagheri, Myra B. Cohen, and Shane Farritor. 2018. Modeling and testing a family of surgical robots: an experience report. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 785–790. https://doi.org/10.1145/3236024.3275534

[26] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. 2011. Unifying execution of imperative and declarative code. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. 511–520. https://doi.org/10.1145/1985793.1985863

[27] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2018. Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation. *ACM Trans. Auton. Adapt. Syst.* 13, 1, Article 3 (April 2018), 36 pages. https://doi.org/10.1145/3149180

[28] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (mar 2015), 66–73. https://doi.org/10.1145/2699417

[29] Jaideep Nijjar and Tevfik Bultan. 2011. Bounded verification of Ruby on Rails data models. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 67–77. https://doi.org/10.1145/2001420.2001429

[30] Haiyu Pan, Yongming Li, Yongzhi Cao, and Zhanyou Ma. 2015. Model checking fuzzy computation tree logic. *Fuzzy Sets Syst.* 262 (2015), 60–77. https://doi.org/10.1016/j.fss.2014.07.008

[31] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. 2020. Towards making formal methods normal: meeting developers where they are. *CoRR* abs/2010.16345 (2020). arXiv:2010.16345 https://arxiv.org/abs/2010.16345

[32] Alireza Sadeghi, Reyhaneh Jabbarvand, Negar Ghorbani, Hamid Bagheri, and Sam Malek. 2018. A temporal permission analysis and enforcement framework for Android. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 846–857. https://doi.org/10.1145/3180155.3180172

[33] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *IEEE Computer* 29, 2 (1996), 38–47. https://doi.org/10.1109/2.485845

[34] Niklas Sorensson and Niklas Een. 2005. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT* 2005, 53 (2005), 1–2.

[35] Clay Stevens, Mohannad Alhanahnah, Qiben Yan, and Hamid Bagheri. 2020. Comparing formal models of IoT app coordination analysis. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment*. 3–10.

[36] Clay Stevens and Hamid Bagheri. 2020. Reducing run-time adaptation space via analysis of possible utility bounds. In *42nd International Conference on Software Engineering, ICSE '20, Virtual Event, USA, July 6-11, 2020*. ACM.

[37] Clay Stevens and Hamid Bagheri. 2022. Combining solution reuse and bound tightening for efficient analysis of evolving systems. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 89–100. https://doi.org/10.1145/3533767.3534399

[38] Clay Stevens and Hamid Bagheri. 2023. Project Website. https://sites.google.com/view/relational-bound-propagation/home.

[39] Maurice H. ter Beek, Kim G. Larsen, Dejan Nickovic, and Tim A. C. Willemse. 2022. Formal methods and tools for industrial critical systems. *Int. J. Softw. Tools Technol. Transf.* 24, 3 (2022), 325–330. https://doi.org/10.1007/s10009-022-00660-4

[40] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 632–647. https://doi.org/10.1007/978-3-540-71209-1_49

[41] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. 2018. Systematic Generation of Non-equivalent Expressions for Relational Algebra. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10817)*, Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl (Eds.). Springer, 105–120. https://doi.org/10.1007/978-3-319-91271-4_8

[42] Wenxi Wang, Kaiyuan Wang, Mengshi Zhang, and Sarfraz Khurshid. 2019. Learning to Optimize the Alloy Analyzer. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 228–239. https://doi.org/10.1109/ICST.2019.00031