Combining Solution Reuse and Bound Tightening for Efficient Analysis of Evolving Systems

Clay Stevens
University of Nebraska-Lincoln
School of Computing
Lincoln, Nebraska, USA
clay.stevens@huskers.unl.edu

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

ABSTRACT

Software engineers have long employed formal verification to ensure the safety and validity of their system designs. As the system changes—often via predictable, domain-specific operations—their models must also change, requiring system designers to repeatedly execute the same formal verification on similar system models. State-of-the-art formal verification techniques can be expensive at scale, the cost of which is multiplied by repeated analysis. This paper presents a novel analysis technique—implemented in a tool called SoRBoT—which can automatically determine domain-specific optimizations that can dramatically reduce the cost of repeatedly analyzing evolving systems. Different from all prior approaches, which focus on either tightening the bounds for analysis or reusing all or part of prior solutions, SoRBoT's automated derivation of domain-specific optimizations combines the benefits of both solution reuse and bound tightening while avoiding the main pitfalls of each. We experimentally evaluate SoRBoT against state-of-the-art techniques for verifying evolving specifications, demonstrating that SoRBoT substantially exceeds the run-time performance of those state-of-the-art techniques while introducing only a negligible overhead, in contrast to the expensive additional computations required by the state-of-the-art verification techniques.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Security and privacy** → *Logic and verification*.

KEYWORDS

formal analysis, bounded verification, speculative analysis

ACM Reference Format:
Clay Stevens and Hamid Bagheri. 2022. Combining Solution Reuse and Bound Tightening for Efficient Analysis of Evolving Systems. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534399

1 INTRODUCTION

Formal verification of software has long been a vital part of engineering dependable, safe software systems [2, 3, 6, 12, 20, 24, 27, 34, 40]. Performing such analyses, however, is often an expensive endeavor, facing scalability issues, particularly for large and complex software systems like those that drive our modern world. Software systems also change frequently over their lifecycle in response to changes in configuration, routine maintenance, and user-initiated operations (to name a few reasons), requiring repeated analysis and thus repeated payment of the costs of formal analysis. In today's fast-moving software development environment, the high cost of formal verification may be prohibitive for rapidly evolving systems, despite the benefits it can offer for reliability, safety, and security.

Many researchers are actively seeking to improve the scalability and adoption of formal verification techniques, including applying those techniques to evolving systems. In particular, using bounded model checking to analyze evolving software specifications and changing configurations (e.g., for self-adaptive systems) has been an active research topic in recent years [4, 7, 8, 14, 28, 31, 37, 39, 41]. Many of the proposed techniques approach the problem in one of two ways: (a) finding ways to incrementally reuse portions of prior solutions [5, 37, 39, 41] or (b) tightening the bounds of the analysis to reduce the search space [4, 8, 16, 22]. Both general approaches serve to limit the work done by the solver with each iteration, but each of the specific realizations comes with its own limitations.

The approaches that seek to reuse portions of the solutions often operate on low-level, syntactic cues or expensive caching techniques to find portions of the formula that can be reused; if there is any deviation from the selected key—however irrelevant at a higher level—the opportunity for reuse is lost. Similarly, tightening the bounds of the analysis has to date required existing domain expertise, as is the case with the approach proposed by Bagheri, et al. [8] to tighten the bound specifically for repeated analysis of Android inter-component communication vulnerabilities. It also requires a method of determining the newer, tighter bounds, which can itself be an expensive operation.

In this paper, we propose a novel approach for efficient repeated formal verification of evolving relational specifications that combines the benefits of both solution reuse and bound tightening while avoiding the main pitfalls of each. We realize our approach in an accompanying tool suite, called SoRBoT for **so**lution **r**euse and **bo**und **t**ightening. Our approach automatically derives high-level opportunities to tighten the bounds used during repeated analysis of an evolving system by reusing portions of prior analysis results. This can dramatically reduce the size of the search space for each subsequent analysis, promising to make large-scale repeated analyses significantly more tractable. The hypothesis guiding this research is that by examining how the application of *operations* permissible in each domain impact the exploration space of bounded verification, it is possible to infer optimizations specific to that specification, facilitating more efficient analyses. SoRBoT performs a one time bounded *speculative analysis*—so called by analogy to speculative execution [19]—on formal definitions of the operations that may be applied to the analyzed system, automatically determining ways to tighten the bounds of analysis, without demanding domain expertise or excessive overhead. SoRBoT then applies the tighter bounds when analyzing the result of applying each operation, greatly reducing the search space for subsequent, repeated formal verification within the target domain. To demonstrate the power of SoRBoT, we evaluate it on a variety of evolving specifications from multiple domains, and compare it against state-of-the-art formal verification techniques. Our experiments show SoRBoT reduces the search space explored during repeated analysis (by more than 74% on average, and by >99.5% for the largest subjects), requires less total time than competing approaches, and adds very little overhead (around 9.25 seconds on average and less than 20 seconds for the largest specifications).

To summarize, this paper makes the following contributions:

- *Automated discovery of opportunities to tighten analysis bounds via solution reuse:* We introduce a novel approach to automatically identify opportunities to both (a) reuse solutions from prior verifications and (b) reduce the search space for formal analysis by tightening the analysis bounds, without relying on costly overhead processing or prior domain-knowledge.
- *SoRBoT implementation:* We realize the presented approach to automatically derive domain-specific optimizations in a tool called SoRBoT. We make SoRBoT available to the research and education community [32].
- *Experiments:* We present empirical evidence of the efficiency gains compared to state-of-the-art formal verification tools using real-world specifications adapted from prior research.

The following section provides a short, illustrative example to describe the benefits of SoRBoT, followed by a more detailed description of our technique. Section 4 presents our empirical evaluation of SoRBoT, and Section 5 discusses the implications of those results. We close with a discussion of threats to validity, prior work, and our conclusions.

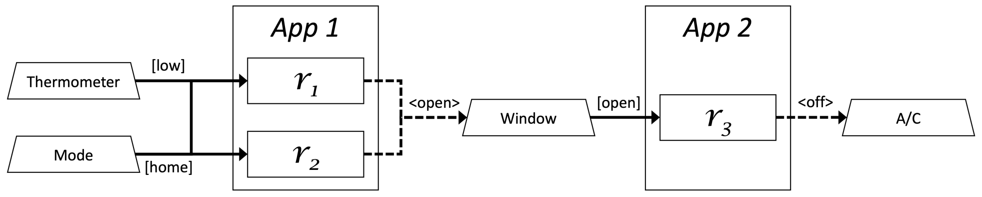
2 ILLUSTRATIVE EXAMPLE

Our approach is best illustrated with an example drawn from a real-world problem: analyzing the safety and security of co-installed IoT apps. IoT ecosystems have experienced a rapid proliferation on the market in recent years, and tools for safety/security analysis of third-party apps available within those ecosystems have been quick to follow [1, 10, 11, 30, 36, 38]. Many of these state-of-the-art analysis tools for IoT apps (e.g., [1, 36]) use formal analysis techniques to find insecure, risky, or inefficient *interactions* among co-installed apps. Such formal analysis techniques rely on a specification of the ecosystem, the apps installed within it, and the interaction rules defined within each app. The set of installed apps (and thus the interaction rules thereof) frequently change as the user installs new apps or updates/removes existing apps over time, thus altering the specification of the system as a whole. As each new app is added or old app removed, the existing security analysis tools mentioned above would need to repeat the full analysis of the (now updated) specification in order to determine if any potential security threats had been introduced or revealed.

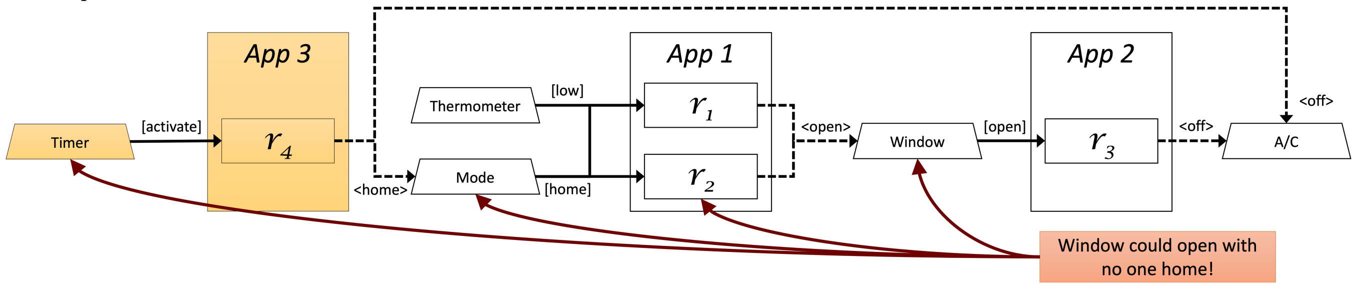
With SoRBoT, information from prior analyses can be reused, allowing for much more efficient repeated analyses of the results of the operations (e.g., adding, updating, or removing apps). Furthermore, SoRBoT automatically discovers which information can be reused by performing an efficient, one-time analysis of a specification of the system and any allowed operations. This removes the need for domain expertise in discovering such optimizations, allowing the tool to improve analysis performance in any domain.

To provide a concrete example, consider a smart home IoT installation such as that depicted in Figure 1. The user has initially installed two apps (cf. Figure 1(a)), each of which are benign. *App 1* includes two rules for interacting with the windows and an external thermometer: the app opens the window if (r_1) the outside temperature drops below a configured threshold and the system is in "home" mode; or (r_2) the system is changed to "home" mode and the outside temperature is currently below the threshold. *App 2* provides only one rule (r_3), which observes pressure sensors on the windows and disables the air conditioning if any windows are open. An analysis of these two apps would discover that there is an interaction chain between these two apps (r_1 or r_2 could trigger activation of r_3), but the safety of the home is not compromised. The extra conditions on r_1 and r_2 ensure the windows stay closed when the user is not at home. Next, assume that the user installs a new app, *App 3*, with a rule (r_4) which periodically shuts off the A/C while the system is in "away" mode to save energy. Unbeknownst to the user, r_4 includes an additional instruction which changes the mode to "home" during the interval where the A/C is disabled. This creates a new link in the interaction chain (r_4 can activate r_2) which *does* have potentially unsafe consequences (the windows may open when the user is away).

Existing IoT analysis tools would need to re-analyze the entire specification to discover the new interaction, even though many parts of the specification are not changed by adding a new app. For example, rules r_1, r_2, and r_3 are still present and unchanged, as are the interaction chains between r_1–r_3 and r_2–r_3. Even so, existing analysis techniques would still consider and explore potential solutions where r_1–r_3 was no longer a member of the "chain" relation defining the interaction chains present in the system. SoRBoT, on the other hand, can automatically discover optimizations that would rule out such (invalid) solutions before performing the analysis. SoRBoT would perform a one-time *speculative analysis* (a novel, exhaustive, yet bounded analysis, detailed in Section 3) of each operation type (e.g., installing a new app) to determine which parts of the specification will remain unchanged after applying that type of operation. It can then optimize subsequent analysis of the results of applying that operation by *fixing* the analysis bounds for unimpacted relations in the specification. For the above example, SoRBoT would discover that no member of the "chain" relation is removed when applying "install app"; therefore, the *lower* bound—the set of items which *must* be members of a given relation from the specification—can be fixed for that relation under the application of that operation. The *upper* bound for "chain"—the set of items



(a) Example smart home IoT installation with two installed apps. *App 1*'s rules (r_1 and r_2) command the window to open if the thermometer is "low" and the mode is "home". *App 2* (r_3) turns off the A/C if the window is open. Interactions (i.e., r_1/r_2 can trigger r_3 by opening the window) are benign, as the window cannot open unless the user is home.



(b) Example application of "install app" operation. When *App 3* is added, the interaction is no longer safe. r_4 is triggered on a timer, and changes the mode to "home". That can trigger r_1/r_2 unexpectedly, opening the window when the user is not at home.

Figure 1: Example of an evolving IoT system. Security analysis of the system in (a) would show there are no vulnerabilities. Applying an operation ("install app") changes the system to configuration (b), which *does* contain a vulnerability. State-of-the-art security analysis techniques must re-analyze the entire configuration; SoRBoT can automatically reuse the unchanged parts of the configuration, and need only analyze the added components (i.e., Timer and App 3).

which *may* be members of the relation—cannot be fixed, as adding a new app may add new interaction chains (e.g., r_4–r_2). By fixing the lower bound, SoRBoT can trim a large portion of the search space before the analysis begins, saving significant analysis time compared to state-of-the-art analysis techniques.

3 APPROACH

This section presents our approach—realized in our tool SoRBoT, for **so**lution **r**euse and **bo**und **t**ightening—that allows for scalable, repeated verification of evolving specifications. Figure 2 shows a simplified, schematic view of the proposed optimization derivation approach. The key innovation of SoRBoT is to automatically discover opportunities to reuse portions of prior solutions to tighten the bounds such that subsequent bounded verification does *not* have to explore the portions of the search space irrelevant to the result of the current operation. In the schematic, the original bounds are represented by the full cube shown in Figure 2a. Figure 2b depicts the values assigned to each variable in a previously generated model instance, which—in conjunction with the results of speculative analysis, Figure 2c—can be reused to substantially tighten the bounds and reduce the search space (Figure 2d).

The core contribution of our approach is a one-time **speculative analysis**, described in more detail in Section 3.1. Given a *relational specification* $\mathcal{S}$ and a set of *domain-specific operations* Λ to be applied to the specification models, SoRBoT can automatically determine which parts of the specification will be affected by subsequent applications of each operation. Later, as the operations are applied (e.g., at runtime, in response to changing requirements), SoRBoT

Listing 1 Example Alloy specification for an abstract IoT ecosystem. Apps are modeled as collections of trigger-condition-action rules operating on uniquely defined device/value pairs.

```
module iot
// IoT ecosystem components
abstract sig App  {
  rules       : set Rule }
abstract sig Rule {
  triggers   : set Device_Value,
  conditions : set Device_Value,
  actions    : set Device_Value }
abstract sig Device_Value { }
// interaction chains
fun chain: Rule -> Rule {
  ((Installed.rules <: (actions.~triggers))
    :> Installed.rules) }
// currently installed apps
sig Installed in App { }
```

can efficiently verify the outcome of applying each operation by reusing previous solutions to tighten the bounds of the analysis (Section 3.2), examining only those parts of the specification that may have changed.

Throughout this section, we will refer to the example IoT system introduced in Section 2 to explain the approach. We present our approach with formal relational specifications expressed using Alloy [17], a language based on first-order relational logic for describing and analyzing software designs. Each Alloy specification

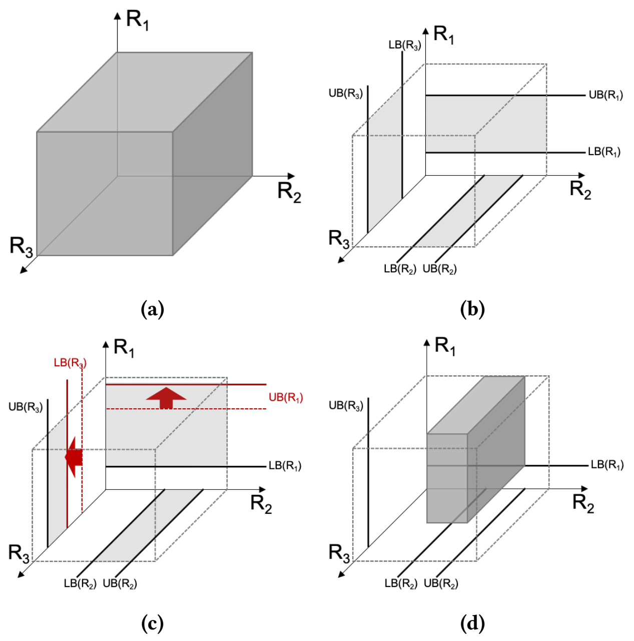


Figure 2: Schematic view of SoRBoT, where the dimensions represent relational variables. For three hypothetical relational variables R1, R2, and R3: (a) default bounds for the original specification; (b) previously computed satisfying instance; (c) results of speculative analysis, where except for the upper bound of R_1, $UB(R_1)$, and the lower bound of R_3, $LB(R_3)$, all other bounds are unaffected by the operation and can be fixed; and (d) adjusted bounds, which are substantially tighter than the original default bounds (Figure 2a).

comprises one or more type-like constructs called *signatures* (keyword `sig`). Each signature defines additional relations among the signatures in the specification as *properties* of the signature. For example, in Listing 1, the `App` signature (Line 3) defines a single binary relation—identified by the `rules` property, Line 4—between the `App` and `Rule` signatures. Relational constraints are represented in Alloy either in `fact` blocks—which must be satisfied to satisfy the specification—or as parameterized functions or predicates (identified with the `fun` keyword) which can be selectively applied under certain conditions. Each signature can be accompanied by a "signature `fact`" block, which acts as a `fact`" applied to each member of that signature. Finally, each Alloy signatures includes a set of *commands* which instruct the Alloy analyzer as to how to interpret the constraints in the signature. The `assert` statement in Listing 4, for example, instructs the analyzer to find instances of the specification satisfying every constraint *except* the one defined in the `assert` statement, presenting *counterexamples* for the assertion which, in this case, represent potentially unsafe app interactions.

Listing 1 shows an Alloy specification of a high-level meta model for describing IoT apps. Lines 3-9 define the components commonly found in IoT apps, modeling apps as collections of trigger-condition-action rules which operate on uniquely-defined device/value pairs. Lines 11-13 define an additional relation which connects two rules

Listing 2 Example Alloy specification for the initial configuration of the IoT system described in Section 2 (see Figure 1a).

```
1  open iot
2  // concrete apps
3  one sig App1 extends App  {} { rules = R1 + R2 }
4  one sig App2 extends App  {} { rules = R3 }
5  // concrete rules
6  one sig R1    extends Rule {} {
7    triggers    = Temp_Lo
8    conditions  = Mode_Home
9    actions     = Window_Open }
10 one sig R2    extends Rule {} {
11   triggers    = Mode_Home
12   conditions  = Temp_Lo
13   actions     = Window_Open }
14 one sig R3    extends Rule {} {
15   triggers    = Window_Open
16   no conditions
17   actions     = AC_Off }
18 // concrete device/value pairs
19 one sig Mode_Away, Mode_Home, Temp_Lo,
20   Window_Open, AC_Off extends Device_Value {}
21 // installed apps
22 fact { Installed = App1 + App2 }
```

Listing 3 Additional Alloy for the updated configuration of the IoT system described in Section 2 (see Figure 1b).

```
1  // new app, rule, and device/value pair
2  one sig App3 extends App  {} { rules = R4 }
3  one sig R4    extends Rule {} {
4    triggers    = Timer_Active
5    conditions  = Mode_Away
6    actions     = Mode_Home }
7  one sig Timer_Active extends Device_Value {}
8  // update the installed apps (replaces Listing 2,
9  // Line 22)
10 fact { Installed = App1 + App2 + App3 }
```

Listing 4 Example Alloy assertion to find an unsafe interaction between IoT apps (i.e., window open while user is away).

```
assert no_open_window { no r1,r2 : Installed.rules {
  r2 in r1.*chain // transitive closure of chain
  Mode_Away   in r1.(triggers + conditions)
  Window_Open in r2.actions }}
```

if the device/value impacted by any of the first rule's actions match the device/value observed in any of a second rule's triggers[1]. Listing 2 defines the specific apps (Lines 3-4), rules (Lines 6-17), and device/value pairs (Lines 19-20) in our IoT example (cf. Figure 1(a)). Line 22 defines the installed apps, completing the specification of the initial configuration of the system. The evolved specification—with the addition of *App 3* and r_4—is expressed in Listing 3 (note that the `fact` block on Line 10 of Listing 3 replaces the similar `fact` on Line 22 of Listing 2). Listing 4 presents a safety property

[1]The relation is specified using this notation to allow for invocation of the transitive closure operator in Listing 4 and to constrain the relation to only consider the rules of installed apps.

Listing 5 Example operation predicates for the IoT example from Section 2. In this notation, term α represents relation α *before* applying the operation, and α' represents the same relation afterward.

```
// installs a new concrete app in the system
pred install_app[a: one App] {
  a not in Installed
  Installed' = Installed + a }
// removes an existing app from the system
pred  remove_app[a: one App] {
  a in Installed
  Installed' = Installed - a }
```

assertion which could be checked to formally verify the safety of the IoT system—in this case, by ensuring that no chain of events could lead to an open window when the user is away. Checking this assertion on the initial specification of the system would find no violations; however, a subsequent analysis including *App 3* would fail, as the chain from r_4 to r_2 violates the assertion.

Lastly, Listing 5 formally defines two abstract operations for the IoT system in Section 2, each of which alters the set of installed applications in the system. These definitions follow the convention that, for any given relation r in the specification, r describes the assignment to that relation prior to the application of the operation, and r' describes the assignment to that relation following application of the operation. Lines 2-3 describe the results of adding a new app, corresponding to the change from Figure 1a to Figure 1b. Lines 5-6 describe the inverse operation, which removes an app.

3.1 Speculative Analysis

The first stage in SoRBoT's process is to determine which if any relations in the relational specification $\mathcal{S}$—denoted as the set $\mathbf{R}$—can be automatically optimized, done by performing a one-time *speculative analysis* of the potential results of applying each operation. We use the term "speculative analysis" by analogy with speculative execution, used for instance in branch prediction to reduce the cost of conditional branch instructions [19] and in recommender systems used in modern integrated development environments [9, 23]. By speculating about future states of a system specification, largely characterized by operations applicable in a given domain, SoRBoT's bounded speculative analysis promises to yield useful information apropos the impacts of each permissible operation on every single relational variable that collectively constitute the specification under analysis.

In specific, SoRBoT seeks to discover any relations $r \in \mathbf{R}$ which have *bounds* that can be fixed (or unchanged) under the application of each operation. The bounds of a given relation, $r \in \mathbf{R}$, are represented by two sets, $\perp_r$ and $\top_r$, which define the set of tuples that *must* be assigned to r in a satisfying model and the set of tuples that *may* be assigned to r in a satisfying model, respectively. The former is called the *lower bound* ($\perp_r$) and the latter is the *upper bound* ($\top_r$). By default, relational model finders (e.g., Kodkod [35]) set the bounds for each relation based on the domains of the relation, using the empty set as the default lower bound and the n-fold Cartesian product of the domains of each relation as the default upper bound. For any given satisfying model instance m of a specification $\mathcal{S}$—denoted $m \models \mathcal{S}$—the tuple assignments to every relation $r \in \mathbf{R}$ within that model (m_r) will thus stand in the following relation to the bounds:

$$m \models \mathcal{S} \implies \forall r \in \mathbf{R} : \perp_r \subseteq m_r \subseteq \top_r \quad (1)$$

Algorithm 1 Speculative analysis algorithm

Input: λ: domain-specific operation
Input: $\mathbf{R}, \mathbf{R'}$: sets of relations before/after applying λ
Output: $R_\perp, R_\top$: sets of relations for which the lower/upper bounds can be fixed

```
1: $R_\perp \leftarrow \varnothing, R_\top \leftarrow \varnothing$
2: for $r \in \mathbf{R}, r' \in \mathbf{R'} : r \prec r'$ do
3:     if $\lambda \implies (r \subseteq r')$ then
4:         $R_\perp \leftarrow R_\perp \cup \{r\}$
5:     if $\lambda \implies (r' \subseteq r)$ then
6:         $R_\top \leftarrow R_\top \cup \{r\}$
```

SoRBoT's speculative analysis leverages that relationship to determine the impact each formally specified operation may have on each of the relations in the specification. First, SoRBoT assumes that the current set of relations, $\mathbf{R}$, represent the relations *before* applying the operation. It then produces a duplicate set of relations, $\mathbf{R'}$, that represent the relations *after* applying the operation, with each corresponding duplicate relation having the same domains and bounds as the original. Formally, SoRBoT defines $\mathbf{R'}$ such that:

$$\forall r \in \mathbf{R} : \exists r' \in \mathbf{R'} : \mathcal{D}_r^N = \mathcal{D}_{r'}^N \wedge \perp_r = \perp_{r'} \wedge \top_r = \top_{r'} \quad (2)$$

We denote the relationship between r and r' as $r \prec r'$.

SoRBoT then "applies" the target operation based on the formal description of the operation and checks the impact on each of the relations. For example, if the predicate describing the operation logically implies that the assignment to a given relation before the operation is a subset of the assignment to that relation after the operation ($r \subseteq r'$), then the *lower bound* for that relation can be counted as *fixed*; the operation will not remove any tuples from a previous satisfying model assignment, so SoRBoT can reuse the assignment to that relation from a prior result as the lower bound for subsequent analysis. If $r \not\subseteq r'$, the default lower bound will be used for r. Similarly, if $r' \subseteq r$, then the *upper* bound can be fixed and reused from a prior analysis result; otherwise, the default upper bound will be used for r. The logical predicate describing an operation is denoted $\lambda \in \Lambda$ in Algorithm 1, which outlines the speculative analysis process for a given operation. Line 2 loops over all the relations in the specification, checking to see if either bound can be fixed and building up a set of relations for which the lower bounds (Lines 3-4) and/or upper bounds (Lines 5-6) can be fixed, respectively. For any relation added by the operation or where $(r \not\subseteq r') \wedge (r' \not\subseteq r)$, the bounds cannot be fixed and the default bounds will be used.

More practically, SoRBoT performs this logical computation using the same underlying technique that drives the overall analysis—bounded model checking. By formulating the conditions on Lines 3 and 5 as safety properties (i.e., properties that we expect to always be true) and checking for a *counterexample*, we can use the underlying model checker efficiently to determine if the conditions hold, up to some desired bound. As this check is still abstract and not tied to any specific system state, the "small-scope hypothesis" would

Listing 6 Example Alloy assertions for finding fixed bounds for IoT operation install_app and chain relation for a scope of 4.

```
check { // lower bound fixed
   all a: set App {
      install_app[a] => chain  in chain'
}} for 4
check { // upper bound fixed
   all a: set App {
      install_app[a] => chain' in chain
}} for 4
```

suggest that only a small bound is required to find a counterexample; however, SoRBoT allows for successive speculative analyses with increasing bounds to improve confidence in its results. As a concrete example, Listing 6 represents the checks that SoRBoT generates to analyze the possibility of adjusting bounds for applying install_app as described in our running example from Section 2. SoRBoT **automatically** creates and checks such assertions for *every* operation and *every* relation in the specification, efficiently and automatically determining which assignments can be reused from prior solutions. Applying the assertions in Listing 6 to our running example, SoRBoT would be able to determine that the *lower* bound for chain can be fixed under application of install_app (no interaction chains are removed), but the *upper* bound cannot be fixed, allowing new apps to add new interaction chains. SoRBoT identifies all such bound adjustments automatically, without demanding any domain knowledge from the end-user.

3.2 Solution Reuse and Bound Adjustment

Using the set of fixed upper and lower bounds identified for each operation by the speculative analysis, SoRBoT is then primed for any subsequent formal verification. More specifically, given specification $\mathcal{S}$ as input, SoRBoT automatically identifies the sets of fixed bounds $R_\perp, R_\top \subseteq \mathbf{R}$ for each operation predicate λ. Before running the verification with the underlying model checker, SoRBoT will first examine the fixed bounds and adjust the bounds of the analysis to match the provided model instance, $m \models \mathcal{S}$, accordingly; for relations in $R_\perp$, for instance, SoRBoT will set the lower bound equal to the tuples assigned that relation in the provided model. In other words:

$$\forall r \in R_\perp : \perp_r \leftarrow m_r \tag{3}$$

$$\forall r \in R_\top : \top_r \leftarrow m_r \tag{4}$$

Having thus adjusted the bounds, SoRBoT supplies the original specification and the adjusted bounds to the bounded model checker and returns the results of its verification. The underlying verification process is unchanged, allowing SoRBoT to be applied to analyze any properties or specifications supported by the underlying bounded model checker.

4 EVALUATION

This section presents our experimental evaluation of SoRBoT. Our evaluation addresses the following research questions:

RQ1. How effective is SoRBoT at reducing the search space for post-operation analyses compared to Alloy Analyzer?

RQ2. What is the performance improvement achieved by SoRBoT compared to state-of-the-art techniques for analyzing evolving specifications?

RQ3. How much performance overhead is introduced by SoRBoT's speculative analysis?

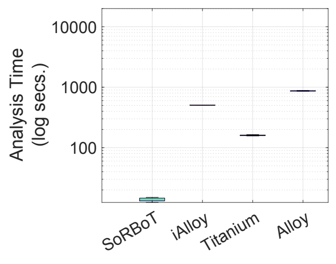
Experimental setup. We conducted the experiments using a custom Java 13 implementation of SoRBoT[2] comprising over 3,800 lines of code. The specifications used in the experiments were developed in Alloy [17] and executed using the Java API of Alloy 5.1, the Kodkod model finder [35] which drives that version of the Alloy Analyzer, and the MiniSAT [29] SAT solver. All experiments were run on an OpenStack instance running Ubuntu 20.04 with 16 2.3GHz VCPUs and 100GB RAM. A maximum 20GB RAM was allowed for the Java heap during each execution.

Measures. During our experiments, we tracked the values of four metrics to use as measures for answering our research questions. First, to determine the scope of the analysis performed by SoRBoT and our baseline systems for RQ1, we tracked the *number of primary variables* in the propositional translation of the relational specification. During translation from the high-level relational specification to the lower level Boolean/SAT formula, model finders, such as Kodkod, assign a Boolean variable to each tuple that is in the upper bound for each relation but *not* in the lower bound for that relation. These variables are used to track whether or not the corresponding tuple was assigned to that relation in the resulting model instance, and are called *primary variables*. The translator may also create auxiliary variables and clauses during translation, but as those tend to depend on the number of primary variables, we report only the primary variables to avoid reporting unnecessary measures. Second, we timed the wall clock running *time taken to verify the properties* included in each subject specification—including any run time overhead—for SoRBoT and all of our baseline systems in order to answer RQ2. To ensure a reasonable total running time for our experiments, each analysis was limited to a total of 24 hours. Lastly, we recorded the wall clock running *time required to perform speculative analysis* using SoRBoT to measure the overhead for RQ3.

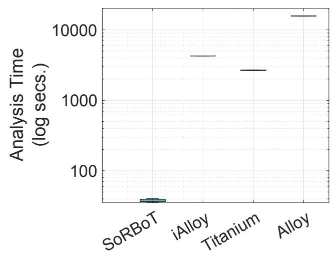
Baselines. For our experimental evaluation, we compare SoRBoT against the latest version of *Alloy Analyzer* (version 5.1) to provide a baseline for our optimizations, as our implementation of SoRBoT extends Alloy 5.1. We also considered three other recent approaches—Titanium [4], Platinum [41], and iAlloy [39]—as baselines for comparison vs. state-of-the-art techniques for verifying evolving specifications.

Titanium. Titanium [4] optimizes repeated analysis of evolving relational specifications by tightening the analysis bounds based on the results from prior analyses. Titanium's method of determining the tighter bounds is very different from SoRBoT's speculative analysis. Rather than running a single small-scope analysis, Titanium *enumerates all satisfying instances* of the base (unmodified) specification and computes *observed bounds* by examining each satisfying instance. These observed bounds are used to tighten bounds for subsequent analysis, although there is no guarantee the observed bound will differ from the default. Enumeration can be incredibly

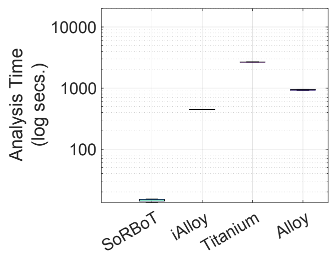
[2]Our custom Java implementation of SoRBoT and our experimental data are available at https://sites.google.com/view/operation-bounder/home



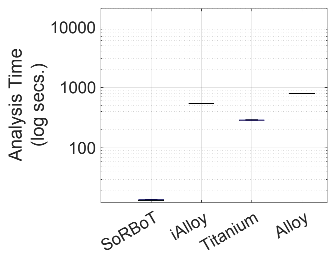
(a) IoT (Bnd. 1)



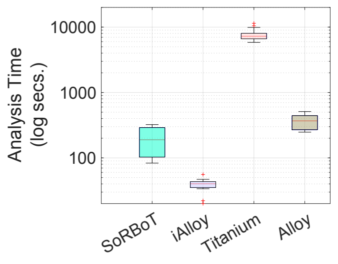
(b) IoT (Bnd. 2)



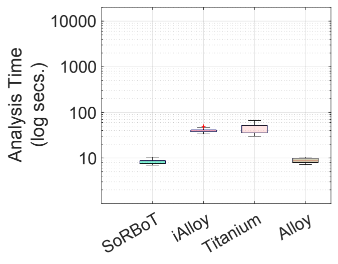
(c) IoT (Bnd. 3)



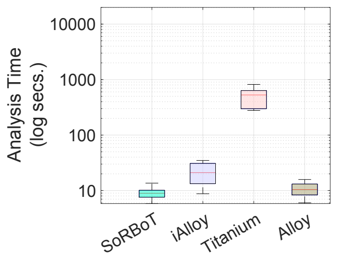
(d) IoT (Bnd. 4)



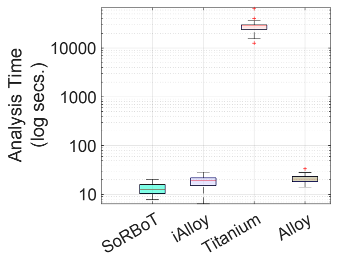
(e) FatFreeCRM (data model)



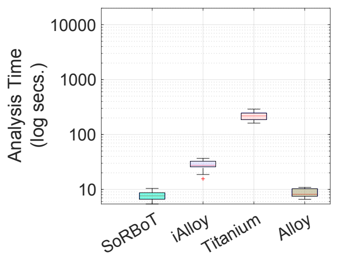
(f) LovdByLess (data model)



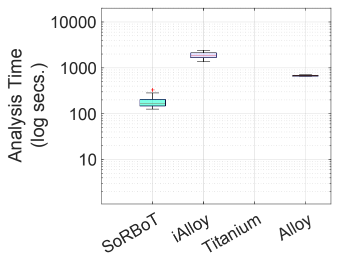
(g) OSR (data model)



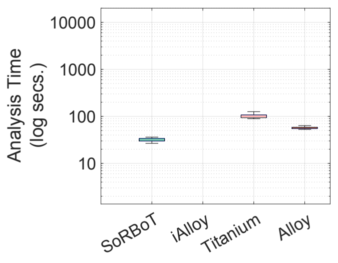
(h) Substruct (data model)



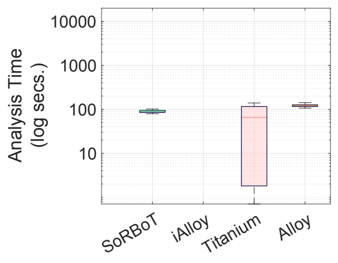
(i) Tracks (data model)



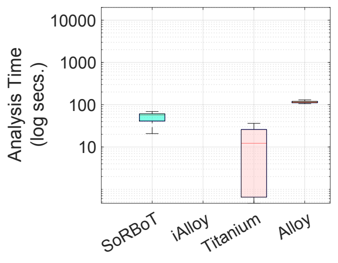
(j) MT-RBAC †



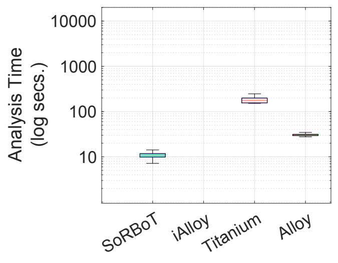
(k) Android (Bnd. 1) ‡



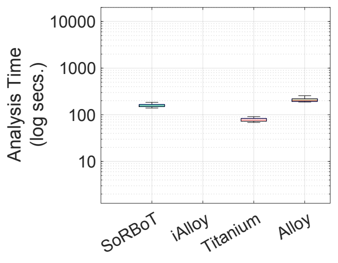
(l) Android (Bnd. 2) ‡



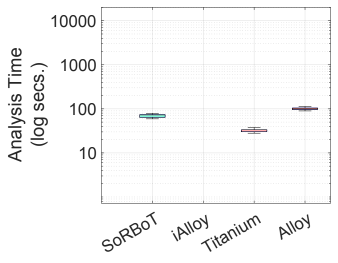
(m) Android (Bnd. 3) ‡



(n) Android (Bnd. 4) ‡



(o) Android (Bnd. 5) ‡



(p) Android (Bnd. 6) ‡

Figure 3: Boxplots showing cumulative analysis time (in seconds, log scale) for applying a sequence of 20 operations and analyzing the results for each operation in the sequence with each analysis technique. Note that the median analysis time taken by SoRBoT across all the subject systems is typically less than the corresponding time taken by the other state-of-the-art techniques. The improvement was more noticeable for larger-scale specifications. † Titanium took longer than 24 hours to finish computing the observed bounds for MT-RBAC. ‡ iAlloy was unable to accurately analyze the changes to the Android specifications (see the third paragraph of Section 4.2)

costly, as there may be an exponential number of satisfying model instances. During our experimental evaluation, we analyzed each of our subject specifications with Titanium both before and after applying each operation to each specification, allowing Titanium to compute the observed bounds on the "before" specification and apply those bounds during the analysis of the "after" specification. When running many operations in sequence, we had Titanium compute the observed bounds after analyzing each resulting specification, as the change to the specification may have changed the observed bounds. Note that to ensure our experimental analyses ran in a reasonable time, we limited Titanium's enumeration while computing the observed bounds to 10,000,000 instances.

Platinum. Platinum [41] also proposes to optimize the analysis of evolving specifications, this time by automatically finding opportunities to reuse solutions for independent clauses in the underlying formula. Platinum works by *slicing* the propositional formula generated by Kodkod [35] (which underpins Alloy Analyzer) into independent clauses—i.e., clauses that share *no* free variables. Similar to Titanium, Platinum also incurs overhead that can grow exponentially as the specification increases in size. While Platinum employs a union-find operation on disjoint set structures to find these slices, the number of variables that must be partitioned can grow exponentially as more relations are added to the specification. Coupled with the additional complexity of determining the weight of each variable for canonicalization (see [41], Section 3.2), the overhead introduced by Platinum can quickly make the approach intractable. When applying operations in sequence, we invoked each operation such that Platinum maintained its cache of previously solved clauses throughout the entire sequence of operations to best demonstrate Platinum's optimizations. For our experiments, we include the time taken to slice the formula and canonicalize the slices as part of the running time for Platinum.

iAlloy. Lastly, iAlloy [39] performs a lexical analysis to find changes to the text of an Alloy specification as it evolves. While the most prominent use case for iAlloy is during the development of a specification, it could also be employed to detect the impacts of executing operations on the specifications of evolving systems. iAlloy works by parsing the text of the Alloy specification in question starting at each command run by the solver. It then generates a dependency graph for the command, finding all the paragraphs of the specification on which the command depends. The approach then generates a checksum for each paragraph, and stores those checksums for comparison with later versions of the specification. If all the dependent checksums match for a given command, then iAlloy reuses the solution for that command and does not invoke the solver. For our experiments, we include the time taken to generate and read dependency graphs and checksums as part of the running time for iAlloy.

Subject systems. We collected a set of 16 system specifications as our experimental subjects, representing evolving specifications in 4 different domains: (1) four bundles of real-world *IoT app* models, all drawn from the suite of models automatically generated by IoTCOM, a recent IoT security analysis technique [1]; (2) five *data models* extracted by a prior study from real-world Ruby-on-Rails systems [25]; (3) an Alloy model of *multi-tenant role-based access control (MT-RBAC)*, as described in [33]; and (4) six bundles of real-world *Android app* models automatically extracted from a pool of 215 Android apps drawn from the security analysis literature [5, 8], including the evaluations of both Titanium and Platinum [4, 41].

All subject specifications—as well as our reference implementation of SoRBoT—are publicly available online for reuse [32].

IoT Security Analysis. First, we evaluated SoRBoT against a suite of models of real-world IoT apps that were automatically captured by IoTCOM, a state-of-the-art IoT security analysis approach [1]. We generated four bundles of individual application models representing a suite of apps that could be installed on the same IoT system. The analysis determines if each suite of co-located apps is subject to risky interactions among the apps. We analyzed the application of two operations—installing a new app or removing an existing app (as described in Listing 5). Each bundle contained up to 10 apps. For the experiment, we started the analysis with a list of one random installed app, and executed the install operation on another random app until all 10 apps were counted as installed. We then randomly removed an installed app until the list of installed apps was empty, analyzing the results of each operation. We have handled uncontrollable factors in the experiments by repeating the experiments 10 times with each analysis technique and report the analysis time for each iteration.

Data models. A system's data model frequently changes as the design of the system evolves. To evaluate SoRBoT's ability to optimize the analysis of such changes, we analyzed the application of common data model transformations on formal specifications of five real-world data models adapted from those extracted by Nijjar and Bultan [25] and packaged with iDaVer [26]. We applied six different types of mutation operations to the data model specifications: (1) changing an association from one-to-one to one-to-many; (2) changing from one-to-many to one-to-one; (3) changing from one-to-many to many-to-many; (4) changing from many-to-many to one-to-many; (5) changing from many-to-many to a pair of one–to-many associations joined by a join table; and (6) changing a join table into a direct many-to-many association. For our experiments, we generated 20 sequences of 20 operations drawn from those six and applied them in sequence to each original data model specification, recording the metrics during the analysis of the properties defined in the original specifications for each data model.

MT-RBAC. User access control (UAC) administration is perhaps the most prominent example of a system where *user-initiated* operations change the system's state. Administrators must be highly trusted within the organizations using these UAC systems. The policies governing access control are often external to the system, with no ability to verify that changes to the system satisfy the governing policies. The lack of any formal validation makes changing user access contentious and time-consuming, as administrators cannot be sure of the ramifications of each operation. Further, administration of user access via third-party, web-based portals (e.g., Azure's portal [21]) requires online analysis of each operation's result, which has to date been out of reach for formal analysis of UAC due to poor scalability [13, 15, 18]. To evaluate SoRBoT's applicability to user-initiated system evolution, we created a formal model of a *multi-tenant role based access control (MT-RBAC)* system as described by Jha, et al. [18]. In an MT-RBAC system, *users* are granted *roles*, which represent bundles of *permissions* granting access to certain resources. Multiple *tenants* within the system can establish *trust* relations among tenants that allows for sharing of roles. In the

Table 1: Mean number of primary variables in the CNF translation when analyzing operations on each subject specification for SoRBoT vs. Alloy, as well as the mean pairwise reduction in the number of primary variables (%).

Subject specification		# Vars (Alloy)	# Vars (SoRBoT)	% Reduction
IoT	Bund. 1	7,278	24	99.7%
	Bund. 2	12,923	27	99.9%
	Bund. 3	6,021	28	99.5%
	Bund. 4	6,947	19	99.7%
Data Models	FatFreeCRM	479	237	50.3%
	LovdByLess	23	17	5.0%
	OSR	78	37	53.3%
	Substruct	84	41	51.9%
	Tracks	66	33	49.4%
Security	RBAC	20,850	5,898	71.2%
Android	Bund. 1	565,670	729	99.9%
	Bund. 2	466,268	550	99.9%
	Bund. 3	230,078	368	99.8%
	Bund. 4	773,569	761	99.9%
	Bund. 5	511,791	584	99.9%
	Bund. 6	374,291	531	99.9%

experiments, we considered the following operations that could be applied by a user during the administration of a role-based access control system: (1) granting/revoking a role to/from a user; (2) granting/revoking a role to/from a permission; (3) adding/removing a relation in a role hierarchy; and (4) adding/removing a trust relation between two tenants of the multi-tenant system. As with the data model subjects, we generated 20 random sequences each with 20 operations and applied each sequence to the base MT-RBAC specification.

Android Security Analysis. Lastly, we evaluated SoRBoT against a suite of formal specifications derived from real-world Android apps that have been used in the evaluation of a number of other recent papers on analysis of evolving specifications [4, 8, 41]. To generate the apps specifications, we used the Covert security analysis tool [5] to produce six bundles of Android app specifications representing a suite of apps that could be installed on the same device. The security analysis aims at determining if each suite of co-located apps is subject to a vulnerability in the communication among the components of each app, such as a privilege escalation attack [5]. We generated 6 bundles containing a random selection of 25 apps drawn from a pool of 215 possible app models. For the experiment, we started the analysis with a list of one random installed app, and executed the add operation on another random app until all 25 apps were counted as installed. We then randomly removed an installed app until the list of installed apps was empty, analyzing the results of each operation.

4.1 RQ1: Improvements in Practice

Table 1 summarizes the size of propositional formulas generated by each of the two techniques, i.e., SoRBoT and Alloy Analyzer, in terms of the number of primary variables in the translation of each specification. The table includes the mean number of primary variables in each translation of the corresponding specification across all experimental runs; for the IoT, data model, and MT-RBAC subjects—which had more operations to test—N=400 each whereas for the Android specifications N=288, for both Alloy and SoRBoT.

As shown, the number of variables in formulas generated by SoRBoT is significantly less than those generated by the Alloy Analyzer. On average, SoRBoT exhibits more than 74% reduction in the size of the translated propositional formulas, compared to those produced by the Alloy Analyzer. The effects of SoRBoT's optimization are clearly visible for more extensive specifications. According to the experimental results, the specifications of the Android app bundles, representing the largest experimental subjects, enjoyed the greatest reduction in variables, with over 99.9% of primary variables removed by SoRBoT. This result clearly shows the effectiveness of our algorithm in reducing the exploration space.

4.2 RQ2: Analysis Time

To address the second research question, we tracked the total time required for each system to analyze the results of applying operations to each of the specifications. For the data models, we generated 20 random sequences of 20 operations each, successively generating updated versions starting from the unmodified base specification. We attempted to analyze each version of the specification with SoRBoT, Alloy Analyzer, Titanium, Platinum, and iAlloy to collect their analysis results. The results for SoRBoT, Titanium, iAlloy, and Alloy Analyzer for our subject specifications are reported in Figure 3, but we were unable to compile results using Platinum.

For Platinum, we used the implementation made available on the website accompanying the published paper. The implementation was unable to finish any of our experiments within the 24-hour limit, spending an excessive amount of time in the lookup phase of the union-find algorithm employed to canonicalize the formulas and ultimately running out of memory on the larger specifications (20GB were allocated for each execution).

We are also unable to report performance results for iAlloy with respect to the Android subject systems as iAlloy provides incorrect analysis results for Android subject systems. Android app specifications include signatures representing concrete apps and components, defined as subsignatures of more abstract signatures. The facts and commands in the specification are defined upon the abstract signatures, which remained unchanged. iAlloy analyzes the text of the specification starting from the executed commands, working back to find all dependencies. As the commands do not explicitly depend on the added/removed subsignatures, iAlloy incorrectly determined no changes were made, and reused existing solutions rather than re-running the analysis, leading to wrong analysis results. Therefore, we exclude the results for the Android specifications as they would not be representative of the time required for a correct analysis. We report iAlloy's results for all other subjects, which did not suffer from the same limitation.

Figure 3 shows box plots of the cumulative analysis times (on a logarithmic scale) for each subject system, grouped by the analysis tools. For all the subject systems, the analysis time required by SoRBoT was less than the time required by Alloy Analyzer to

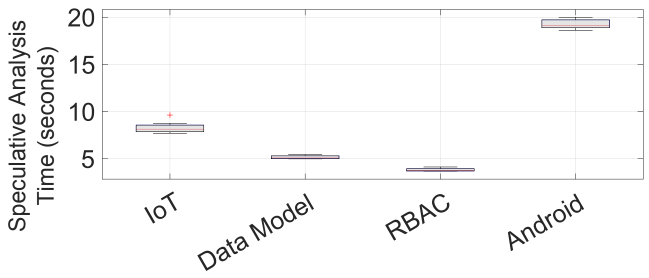


Figure 4: Box plots showing the running time (in seconds) for speculative analysis with SoRBoT for each of the four base subject specifications (N=10).

analyze the same specification. The improvement was more noticeable for larger scale specifications, e.g., IoT app bundles. SoRBoT also outperformed Titanium in terms of the analysis time across all subject systems. While in a few Android cases, Titanium performed comparably to SoRBoT, in all other cases, SoRBoT remarkably outperformed Titanium. In one case (MT-RBAC), Titanium was not able to finish computing the observed bounds for the first operation within the 24-hour time limit.

4.3 RQ3: Overhead

To measure overhead, we executed only the speculative analysis for each of our subject systems 10 times, with the results depicted as box plots in Figure 4. According to the experimental results, the execution time overhead incurred by the speculative analysis to automatically derive optimized analysis bounds is 9.25 seconds on average, with no speculative analysis taking longer than 20 seconds. The overhead of the one-time speculative analysis for each domain is negligible considering the actual analysis time for the specifications in each respective domain. Note that the speculative analysis is operating on high-level *meta models* for specifications to automatically identify opportunities for bound adjustments. Therefore, all four IoT app bundles, all five data models, and all six Android app bundles share the same fixed bounds with the other specifications in their domain. The Android analysis took the longest, in keeping with the fact that the meta model describing Android intercomponent communication is much larger than the entity-relation meta model for the data models and the model for MT-RBAC, with the IoT app bundles only slightly larger. As the speculative analysis only needs to be performed *once* for each domain, the overhead reported here would also be amortized across every subsequent run-time verification.

5 DISCUSSION

Overall, we interpret the experimental results to show that SoRBoT provides a practical and efficient alternative to state-of-the-art techniques for analysis of evolving specifications, without suffering from the drawbacks of the existing approaches. As shown in Table 1, SoRBoT cuts the scope of analysis in half (often far better) for all but the smallest subject systems, showing a promising reduction in the amount of work the solver must do for *each* run-time operation verification. The reductions in the total analysis time vs. the Alloy baseline also benefit from the same underlying fact: the improvements shown by SoRBoT accrue *with each repeated analysis*, resulting in an increasing improvement each time an operation is formally verified.

This comes at the cost of a very low overhead; as shown in Figure 4, The speculative analysis took around 8, 5, 4, and 20 seconds for the IoT, data model, MT-RBAC, and Android specifications, respectively. In the case of MT-RBAC and Android, this overhead accounts for a small fraction of the total analysis time saved *for each repeated operation*, despite accruing exactly once. Compared to Titanium SoRBoT introduces negligible overhead, and Platinum required so much preprocessing that it could not be included in our reported results. SoRBoT demonstrates a remarkable improvement over the state-of-the-art techniques. As shown in Listing 5, SoRBoT can automatically derive impressive optimizations from even very simple operation descriptions.

In terms of analysis time, Titanium performed poorly for most subject systems. This was due to the extreme cost of enumerating all satisfying models to compute the observed bounds, even when limited to 10,000,000 instances. As SoRBoT requires no explicit enumeration, it does not suffer from that drawback. Titanium was able to compare favorably with SoRBoT in the Android domain, the domain against which Titanium was originally evaluated in [4]. Also, Titanium showed a vast spread of analysis times for two Android bundles; this may be due to the higher number of solutions enumerated by the Titanium approach. In cases where Titanium's analysis returns no instances after applying an operation, Titanium has no instances to enumerate to observe the bounds and thus must translate the *entire* specification for the subsequent operation. This leads to large swings in translation time for the Android specifications, which require a great deal of translation time when unadjusted. SoRBoT does not have this limitation, as its bound adjustment does not require enumeration. iAlloy performed well on the smaller specifications, but required more time for the large IoT specifications. This may be due to the extra overhead of having to build the dependency graph and compute checksums on the paragraphs of the specification. Notably, iAlloy outperformed the Alloy baseline for all subject systems, but required more (median) analysis time than was required by SoRBoT.

6 THREATS TO VALIDITY

The main threat to the internal validity of our experiment is the fidelity of our implementation and the correctness of our generated model variants. The entire system—both our Java implementation and our experimental scripts—underwent intensive testing during development to ensure they were correctly executing as desired and reporting the correct measurements during the experiments. To further validate the consistency of results produced by SoRBoT's optimized analysis with those produced by the Alloy Analyzer, we compared the solutions produced by the two techniques. The experimental results confirm that SoRBoT computes the same set of solutions as the Alloy Analyzer in all cases, corroborating our theoretical expectation. To ensure our external validity, we drew our subject specifications from four disparate domains representing active areas of research in the verification of evolving specifications. All of our subject specifications were drawn directly from the literature, with small alterations to allow our tool to analyze the results of the operations and synthesize the necessary variants to test changes to the model. Lastly, the measures we chose for our experiment are well established and provide no threat to the validity of our construct.

7 RELATED WORK

Formal verification of evolving specifications has recently been a very active area of research [4, 8, 14, 28, 37, 39, 41]. In particular, approaches that use bounded model checking have been very popular, tending to approach solutions to the problem from two main directions. Approaches such as Titanium [4] and Flair [8] introduce and validate the method of *tightening bounds* based on the instances produced by bounded model checking. However, as discussed in Section 5, both rely on enumerating satisfying models of at least one specification to compute the observed bounds. Also, the latter relies on existing domain expertise that must be provided by the user about which relations are affected by each operation. For Titanium, the altered model must constitute a *specialization* of the original such that all instances satisfying the modified solution must satisfy the original. This limits the generality of their approach. SoRBoT does not suffer that limitation, as any domain-specific operation can be analyzed. On the other hand, Flair analyzes a specific pair of operations in a single domain–namely adding or removing apps from a bundle of apps installed on an Android phone. Flair's developers leverage their expertise in the domain to specify the bounds that can be fixed for each operation. As demonstrated in Section 4, SoRBoT not only automatically detects those bounds when applied to the Android domain, but can derive such domain-specific optimizations for *any* other domain.

Other approaches tend to focus on *solution reuse*, finding opportunities to reuse all or part of prior solutions to reduce the scope of subsequent analyses [37, 39, 41]. Platinum [41] extends Alloy Analyzer by slicing the propositional formula translated via Kodkod into independent subclauses and caching the results for each subclause for future use. This introduces an additional computational burden for the analysis of large specifications, as each verification must also slice and canonicalize the formula as well as exploring and updating the cache with any new subclauses. This approach is similar to recent approaches derived from Green, proposed by Visser, et al. [37], which also stores and reuses portions of the propositional formula. iAlloy—proposed by Wang, et al. [39]—approaches the reuse in a different way by storing and reusing solutions wholesale when verifying multiple disjoint properties of a given specification. iAlloy performs a lexical analysis of the specification to determine which "run" or "check" commands are impacted by the change, and re-uses prior solutions for any commands that were not altered. Titanium, Platinum, and iAlloy were used as baselines for our experiments, as described in Section 4.

Different from all these techniques, SoRBoT explores the possibility of automated discovery of domain-specific optimizations that combines the benefits of both solution reuse and bound tightening while avoiding the main pitfalls of each. To the best of our knowledge, SoRBoT is the first approach that seeks to automatically derive domain-specific optimization without direct input from domain experts.

8 CONCLUSION

In this paper, we presented SoRBoT, a novel approach to automatically discover and exploit potential domain-specific optimizations for repeated verification of evolving formal specifications. SoRBoT performs an inexpensive, one-time *speculative analysis* to determine which if any relational bounds in the provided relational specification can be "fixed", indicating they can be reused from a prior solution drawn from a related specification. Experimental results indicate that (a) SoRBoT improves upon Alloy Analyzer's (our baseline) by cutting the search space in half or better for nearly all of our subject specifications; (b) speculative analysis introduces negligible overhead; and (c) SoRBoT's run-time analysis time for verifying the results of domain-specific operations improves upon the baseline and is at least on par with state-of-the-art techniques for verifying evolving specifications.

In future research, we would seek to improve the speedups due to SoRBoT by targeting the translation of relational specifications to propositional logic and SAT formulas to see if any similar potential exists to reuse prior translations or limit the scope of the translation. We would also like to explore combinations of the SoRBoT's speculative analysis with other techniques promoting full or partial solution reuse, to see if similar solver-based approaches can automate the discovery of reuse opportunities in other parts of the formal verification process.

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, and CCF-2139845 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] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2018. A formal approach for detection of security flaws in the android permission system. *Formal Aspects Comput.* 30, 5 (2018), 525–544. https://doi.org/10.1007/s00165-017-0445-z

[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*. 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. 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

[7] 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

[8] 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*. 357–368. https://doi.org/10.1109/ICSME.2018.00044

[9] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2010. Speculative analysis: exploring future development states of software. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and Kevin J. Sullivan (Eds.). ACM, 59–64. https://doi.org/10.1145/1882362.1882375

[10] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, Berkeley, CA, USA, 1687–1704.

[11] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 147–158.

[12] 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

[13] Arnaud Dury, Sergiy Boroday, Alexandre Petrenko, and Volkmar Lotz. 2007. Formal Verification of Business Workflows and Role Based Access Control Systems. In *Proceedings of the First International Conference on Emerging Security Information, Systems and Technologies, SECURWARE 2007, October 14-20, 2007, Valencia, Spain*, Lourdes Peñalver, Oana Andreea Dini, Judie Mulholland, and Octavio Nieto-Taladriz (Eds.). IEEE Computer Society, 201–210. https://doi.org/10.1109/SECUREWARE.2007.4385334

[14] Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. 2017. Flexible SAT-based framework for incremental bounded upgrade checking. *Int. J. Softw. Tools Technol. Transf.* 19, 5 (2017), 517–534. https://doi.org/10.1007/s10009-015-0405-y

[15] Anna Lisa Ferrara, P. Madhusudan, Truc L. Nguyen, and Gennaro Parlato. 2014. Vac - Verifier of Administrative Role-Based Access Control Policies. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 184–191. https://doi.org/10.1007/978-3-319-08867-9_12

[16] J.P. Galeotti, N. Rosner, C.G. Lopez Pombo, and M.F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE Transactions on Software Engineering* 39, 9 (Sept. 2013), 1283–1307. https://doi.org/10.1109/TSE.2013.15

[17] Daniel Jackson. 2003. Alloy: A Logical Modelling Language. In *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*. 1. https://doi.org/10.1007/3-540-44880-2_1

[18] 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

[19] Butler W. Lampson. 2008. Lazy and speculative execution in computer systems. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 1–2. https://doi.org/10.1145/1411204.1411205

[20] 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

[21] Microsoft. 2020. Add or remove Azure role assignments using the Azure portal. https://docs.microsoft.com/en-us/azure/role-based-access-control/role-assignments-portal. accessed: 11 Dec 2020.

[22] 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

[23] Kivanç Muslu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2012. Speculative analysis of integrated development environment recommendations. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 669–682. https://doi.org/10.1145/2384616.2384665

[24] 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

[25] 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

[26] Jaideep Nijjar, Tevfik Bultan, and Ivan Bocic. 2020. iDaVer. https://vlab.cs.ucsb.edu/idaver/.

[27] 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

[28] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2015. Incremental Upgrade Checking. In *Validation of Evolving Software*, Hana Chockler, Daniel Kroening, Leonardo Mariani, and Natasha Sharygina (Eds.). Springer, 55–72. https://doi.org/10.1007/978-3-319-10623-6_6

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

[30] 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.

[31] Clay Stevens and Hamid Bagheri. 2020. Reducing run-time adaptation space via analysis of possible utility bounds. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1522–1534. https://doi.org/10.1145/3377811.3380365

[32] Clay Stevens and Hamid Bagheri. 2022. SoRBoT website. https://sites.google.com/view/operation-bounder/home.

[33] Bo Tang, Qi Li, and Ravi Sandhu. 2013. A multi-tenant RBAC model for collaborative cloud services. In *Eleventh Annual International Conference on Privacy, Security and Trust, PST 2013, 10-12 July, 2013, Tarragona, Catalonia, Spain, July 10-12, 2013*, Jordi Castellà-Roca, Josep Domingo-Ferrer, Joaquín García-Alfaro, Ali A. Ghorbani, Christian D. Jensen, Jesús A. Manjón, Iosif-Viorel Onut, Natalia Stakhanova, Vicenç Torra, and Jie Zhang (Eds.). IEEE Computer Society, 229–238. https://doi.org/10.1109/PST.2013.6596058

[34] 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

[35] 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*. 632–647. https://doi.org/10.1007/978-3-540-71209-1_49

[36] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, Guoqing Harry Xu, and Shan Lu. 2020. Understanding and Automatically Detecting Conflicting Interactions between Smart Home IoT Applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1215–1227.

[37] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 58. https://doi.org/10.1145/2393596.2393665

[38] Qi Wang, Wajih Ul Hassan, Adam M. Bates, and Carl A. Gunter. 2018. Fear and Logging in the Internet of Things. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.*

[39] Wenxi Wang, Kaiyuan Wang, Milos Gligoric, and Sarfraz Khurshid. 2019. Incremental Analysis of Evolving Alloy Models. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11427)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 174–191. https://doi.org/10.1007/978-3-030-17462-0_10

[40] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. 2009. Formal methods: Practice and experience. *ACM Comput. Surv.* 41, 4 (2009), 19:1–19:36. https://doi.org/10.1145/1592434.1592436

[41] Guolong Zheng, Hamid Bagheri, Gregg Rothermel, and Jianghao Wang. 2020. Platinum: Reusing Constraint Solutions in Bounded Analysis of Relational Logic. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12076)*, Heike Wehrheim and Jordi Cabot (Eds.). Springer, 29–52. https://doi.org/10.1007/978-3-030-45234-6_2