

Bounded Satisfiability Checking of FOL* Formulas with Aggregations

Abstract. Software systems handling data are increasingly required to comply with legal properties (LPs) aimed at ensuring security and data privacy. Satisfiability checking for first-order logic with quantifiers over relational objects (FOL*) enables expression and detection of LP violations on early system designs. While generally effective, FOL* does not directly support data constraints with aggregated values, which are often found in compliance policies, and that can lead to complex data constraints that negatively affect the efficiency of FOL* satisfiability checking. To enable efficient support, we extend FOL* with aggregation, resulting in a language FOL⁺⁺, and propose a satisfiability checking algorithm, **TOOL**, for FOL⁺⁺ which supports reasoning about aggregation by over- and under-approximating the aggregated values and incrementally refining these approximations to derive the satisfiability result. Running **TOOL** on real-world and academic examples with aggregation from various domains showed that the tool was able to solve many previously intractable problems and provided 19X speed-ups compared to the state-of-the-art satisfiability checker **LEGOS**.

1 Introduction

Software systems handling data are increasingly required to comply with laws and regulatory policies (LPs) aimed at ensuring data privacy. First-order logic with quantifiers over relational objects (FOL*) has been proposed as a unified foundational formalism to capture LPs [21, 22] expressed in MFOTL [10] and SLEEC [43]. Checking FOL* satisfiability [22] enables the detection of LP violations with respect to a system design. Despite the general undecidability of the FOL* satisfiability problem, a practical approach involves checking bounded FOL* satisfiability to establish bounded guarantees for the system’s design: namely, ensuring that no LP violation can occur within a specified limit on the number of data actions.

LEGOS, a state-of-the-art approach for bounded FOL* satisfiability [22], searches for a satisfying solution to FOL* formulas which corresponds to one possible violation of the LP. **LEGOS** determines satisfiability of FOL* formulas by computing their over and under-approximations, solving them, and refining them incrementally until the satisfiability result is soundly derived. While generally effective, **LEGOS** cannot efficiently support data constraints with aggregated values which are often found in compliance policies [9], such as financial and data privacy regulations, because aggregation is not directly supported in FOL*. For instance, consider an LP, which we refer to as P_1 , derived from the article 63 of the European directive on payment services in the internal market [18]. P_1 ,

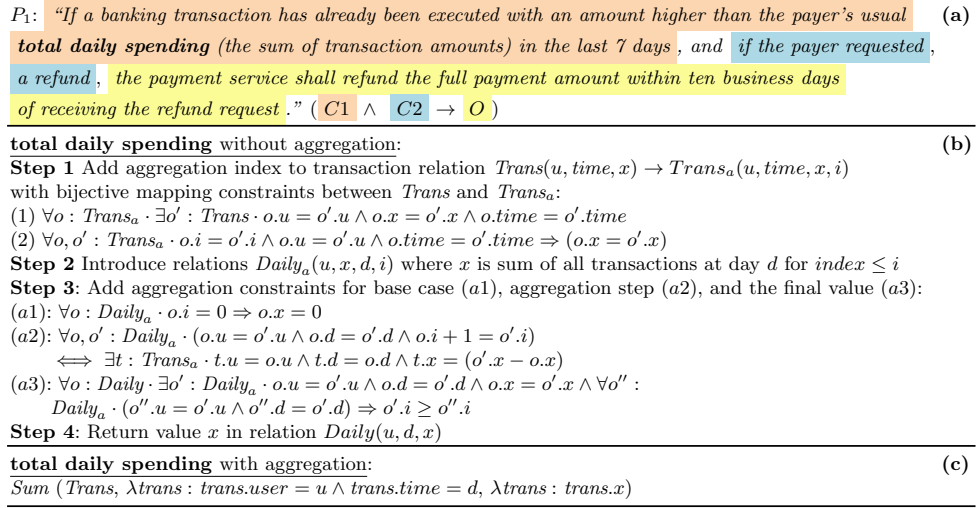


Fig. 1. Several FOL* properties: (a) a legal property P_1 ; a **total daily spending** of a user u on day d defined (b) without aggregation; (c) with aggregation.

shown in Fig. 1(a), includes an obligation when conditions $C1$ and $C2$ are satisfied: condition $C1$ compares a single transaction amount with its payer’s *total daily spending*, which can be expressed as the sum of the payer’s transactions aggregated over a day.

A simple solution to support aggregation in FOL* is to encode its recursive definitions [35] in the logic and solve them with the existing reasoning machinery. For instance, the FOL* model for capturing a payer’s *total daily spending* (without aggregation) is shown in Fig. 1. This model creates a symbolic list (inspired by the concept of a symbolic set [3, 19]) of ordered transactions to be aggregated over, by assigning each one a unique ID. Note that the list must be symbolic because both the number and value of transactions are to be decided. Then, the model specifies the recursive aggregation rule for the sum over the list from left to right. Constraint $a1$ specifies the base case of the summation, as the sum over an empty list is 0. The constraint $a2$ specifies how the value of the sum should be updated for each step (i.e., from index i to $i + 1$). Finally, the constraint $a3$ ensures that the total daily spend is the result of the summation over the entire list (i.e., the sum at the last step). However, the model of aggregation is ineffective for satisfiability checking and often creates complex data constraints since it operates on a symbolic list, where the size and the value of the list are variables. The negative impact of the aggregation model is also acknowledged by the authors of LEGOS [22]. Therefore, there is a real need to supporting aggregation for FOL* satisfiability checking.

In this paper, we introduce **TOOL**, a bounded satisfiability approach capable of handling FOL* extended with aggregation, which we refer to as FOL*⁺. Specifically, **TOOL** receives an FOL*⁺ property and system requirements and incrementally grounds the FOL*⁺ constraints to eliminate the quantifiers by

considering an increasing number of relational objects, and checks the satisfiability of the resulting constraints using an SMT solver. In order to effectively support aggregations, **TOOL** enriches boolean over- and under-approximations (using techniques from **LEGOS**), by computing over- and under-approximations for numerical terms, including the aggregated values appearing in the specification. While the numerical under-approximation is bounded by the domain of relational objects, its over-approximation is bounded symbolically from above and below using a *local upper-bound* and a *global lower-bound*, respectively. The numerical over- (resp. under-) approximations for the aggregated values are refined incrementally and form monotonically decreasing (resp. increasing) sequences converging to a satisfying solution. Running **TOOL** on real-world and academic examples from various domains solved 2.6 times more instances and provided a significant runtime improvement (19X speed-up) compared to **LEGOS**.¹

The remainder of the paper is organized as follows: Sec. 2 recalls the notion of FOL*. Sec.3 introduces the extension of aggregation for FOL*+. Sec. 4 presents our approach for the FOL*+ bounded satisfiability checking, with a focus on support for aggregations. Sec. 5 reports on the experiments we have conducted to validate **TOOL** and compare it to **LEGOS**. Sec. 6 compares **TOOL** to existing satisfiability checking approaches. Finally, Sec. 7 concludes the paper.

2 Preliminaries

In this section, we briefly describe aggregation and FOL* [22], a first-order logic with quantifier over relational objects.

Aggregation. Aggregation has the syntax $A(N)$, where $A \in \{Sum, Count, Max, Min\}$ is an aggregation function and N defines a collection of numerical values. We assume that N is a finite bag (multi-set) of integers. Let $ite(p, a, b)$ denote the “if-then-else” function that takes a Boolean condition p and values a and b and returns a if p is true and b otherwise. The semantics of aggregation for bags is defined recursively as follows:

- (1) $Sum(\emptyset) = 0$, $Sum(N \cup x) = Sum(N) + x$;
- (2) $Count(\emptyset) = 0$, $Count(N \cup x) = Count(N) + 1$;
- (3) $Max(\emptyset) = -\infty$, $Max(N \cup x) = \max(Max(N), x)$; and
- (4) $Min(\emptyset) = \infty$, $Min(N \cup x) = \min(Min(N), x)$.

FOL with relational objects (FOL*). A *signature* S is a tuple (C, R, ι) , where C is a set of constants, R is a set of relation symbols, and $\iota : R \rightarrow \mathbb{N}$ is a function that maps a relation to its arity. We assume that the domain of constant C is \mathbb{Z} , where the theory of linear integer arithmetic (LIA) holds. Let V be a set of variables in the domain \mathbb{Z} . A *relational object* $o : r$ of class $r \in R$ is an object with $\iota(r)$ regular attributes and two special attributes, where every attribute is a variable. We assume that all regular attributes are ordered and denote $o[i]$ to be the i th attribute of o . Some attributes are named, and $o.x$ refers to o ’s attribute with the name ‘ x ’. Each relational object o has two special attributes, $o.ext$ and $o.time$. The former is a boolean variable indicating whether o exists

¹ All source files and case studies are available in [4].

in a solution, and the latter is a variable representing the occurrence time of o . For convenience, we define a function $\text{CLS}(o : r)$ to return the relational object's class r . Let a *term* t be defined inductively as $t : c \mid \text{var} \mid o[i] \mid o.x \mid t + t \mid c \times t$ for any constant $c \in C$, any variable $\text{var} \in V$, any relational object o , any index $i \in [0, \iota(r)]$ and any valid attribute name x . Given a signature S , the syntax of the FOL* formulas is defined as follows: (1) \top and \perp , representing values “true” and “false”; (2) $t = t'$ and $t > t'$, for term t and t' ; (3) $\phi \wedge \psi$, $\neg\phi$ for FOL* formulas ϕ and ψ ; (4) $\exists o : r \cdot (\phi)$ for an FOL* formula ϕ and a class r ; (5) $\forall o : r \cdot (\phi)$ for an FOL* formula ϕ and a class r . The quantifiers for FOL* formulas are limited to relational objects, as shown by rules (4) & (5). Operators \vee and \forall can be defined in FOL* as follows: $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$ and $\forall o : r \cdot \phi = \exists o : r \cdot \neg\phi$. We say an FOL* formula is in a *negation normal form* (NNF) if negations (\neg) do not appear in front of \neg , \wedge , \vee , \exists and \forall . For the rest of the paper, we assume that every FOL* ϕ is in NNF.

Given a signature S , a *domain* D is a finite set of relational objects. An FOL* formula *grounded* in the domain D (denoted by ϕ_D) is a quantifier-free FOL formula that eliminates quantifiers on relational objects using the rules: (1) $\exists o : r \cdot (\phi)$ to $\bigvee_{o' : r}^D (o'.\text{ext} \wedge \phi_D[o \leftarrow o'])$ and (2) $\forall o : r \cdot (\phi)$ to $\bigwedge_{o' : r}^D (o'.\text{ext} \Rightarrow \phi_D[o \leftarrow o'])$ where ϕ_D is the result of grounding ϕ in D . An FOL* formula ϕ is *satisfiable in D* if there exists a variable assignment v that evaluates ϕ_D to \top according to the standard semantics of FOL (with LIA). An FOL* formula ϕ is *satisfiable* if there exists a finite domain D such that ϕ is satisfiable in D . We call $\sigma = (D, v)$ a *satisfying solution* to ϕ , denoted as $\sigma \models \phi$. Given a solution $\sigma = (D, v)$, we say a relational object o is in σ , denoted as $o \in \sigma$, if $o \in D$ and $v(o.\text{ext})$ is true. The *volume of the solution*, denoted as $\text{vol}(\sigma)$, is $|\{o \mid o \in \sigma\}|$.

Example. Let a be a relational object of class A with an attribute name val . The formula $\forall a : A. (\exists a' : A \cdot (a.\text{val} < a'.\text{val}) \wedge \exists a : A \cdot a.\text{val} = 0)$ has no satisfying solutions in any finite domain. On the other hand, the formula $\forall a : A \cdot (\exists a', a'' : A \cdot (a.\text{val} = a'.\text{val} + a''.\text{val})) \wedge \exists a : A \cdot a.\text{val} = 5$ has a solution $\sigma = (D, v)$ of volume 2, with the domain $D = (a_1, a_2)$ and the value function $v(a_1.\text{val} = 5)$, $v(a_2.\text{val} = 0)$ because if $a \leftarrow a_1$ then the formula is satisfied by assigning $a' \leftarrow a_1$, $a'' \leftarrow a_2$; and if $a \leftarrow a_2$, then the formula is satisfied by assigning $a' \leftarrow a_2$, $a'' \leftarrow a_2$.

3 Aggregation and FOL*⁺

In this section, we present FOL*⁺, which extends FOL* with aggregation.

Let S be a class of relational objects, $A \in \{\text{Sum}, \text{Count}, \text{Max}, \text{Min}\}$ be an aggregation function, p be a predicate for relational objects in S , and $\text{val} : S \rightarrow \mathbb{Z}$ be a function that maps a relational object of class S to a numerical value. An *aggregation in FOL*⁺* is a tuple $A(S, p, \text{val})$, and its semantics is defined for a given domain D and a valuation function v as:

$$v(A(S, p, \text{val}), D) := A([v(\text{val}(s), D) \mid s \in S_D \text{ if } v(s.\text{ext} \wedge p(s), D)]),$$

where S_D is the set of unique relational objects of class S in domain D , and $v(expr, D)$ is the result of evaluating the FOL⁺⁺ expression $expr$ with valuation v over domain D . Intuitively, the predicate p and the function val together generate a bag (multi-set) of numerical values (from the set of relational objects S in domain D) where the aggregation function A is applied.

Since p and val are FOL⁺⁺ functions that might return expressions including aggregations, the value of an aggregation might not be decidable, even in a fixed domain. To illustrate this, consider $a = \text{Sum}(S, \lambda s : \top, \lambda s : a)$, where the value function always returns the summation a itself, and the computation of a never terminates. To ensure that the value of the aggregation is always decidable in any given domain D , we assume that all aggregations are *stratified*.

Definition 1 (Stratified Aggregations). *An aggregation $A(S, p, val)$ is stratified at layer 0 if for every $s \in S$, $p(s)$ and $val(s)$ do not contain summations. $A(S, p, val)$ is stratified at layer n if for every $s \in S$, $p(s)$ and $val(s)$ only contain aggregations that are stratified at layer $n - 1$ or lower. Given an FOL⁺⁺ formula ϕ with N unique functions, if $A(S, p, val)$ is an expression in ϕ , then $A(S, p, val)$ is stratified if and only if it is stratified at layer N or below.*

Example: Let $\text{Trans}(u, x)$ be a relation describing a transaction where u is a user who initiates a transaction, and x is the amount involved in the transaction. The relational object of class Trans has four attributes: u , x , ext and $time$. Consider the property P_1 in Fig. 1(a) which contains the condition $C1$ that compares the size of user transactions with their daily spending of the last 7 days. Condition $C1$ can be formalized in FOL⁺⁺ as follows: $C1 := \exists tr : \text{Trans} \cdot (\forall t : \text{Time} \cdot \text{trans.time} - 7 \leq t < \text{trans.time} \Rightarrow \text{Sum}(\text{Trans}, \lambda tr' : tr'.u = tr.u \wedge tr'.time = t, \lambda tr' : tr'.x) < tr.x)$. $C1$ specifies the existence of a relational object tr of class Trans such that the daily spending for the last 7 days ($\forall t : \text{Time} \cdot \text{trans.time} - 7 \leq t < \text{trans.time}$) is less than $tr.x$. For each day t , the total daily expenditure of the user $tr.u$ during t is calculated with a summation that considers all transactions (tr') that are initiated by $tr.u$ ($tr'.u = tr.u$) and occur at time t ($tr'.time = t$). The summation then computes the total transaction amount ($tr'.x$) and compares it with the target value ($\text{Sum}(\dots) < tr.x$). $\text{Sum}(\dots)$ is stratified at layer 0 because the expressions returned by the predicate function (i.e., $tr'.u = tr.u \wedge tr'.time = t$) and the value function (i.e., $tr'.x$) do not contain sum.

For a summation $\text{Sum}(S, p, val)$, without loss of generality, we assume that the range of val is \mathbb{N} . This assumption is made because we can rewrite any summation with a valuation function val ranging over \mathbb{Z} into two summations with valuation functions ranging over \mathbb{N} : $\text{Sum}(S, p, val) = \text{Sum}(S, \lambda s.p(s) \wedge val(s) > 0, val) - \text{Sum}(S, \lambda s.p(s) \wedge val(s) < 0, -val)$. The assumption about the range of val ensures that aggregated sums are monotonically non-decreasing as the domain D expands. The same monotonicity property holds for Max and Count , while the negative monotonicity property holds for Min . In Sec. 4.2, we leverage this monotonicity property to support aggregation for satisfiability checking.

Remark. FOL⁺⁺ captures the monitorable fragment of MFOTL _{ω} [11]. More details are available in Appendix E.

Algorithm 1 SEARCH-A: search for a bounded (by vb) solution to the conjunction of the set of FOL⁺⁺ formula F .

Input A set of FOL⁺⁺ formula $F = \{\phi_1, \dots, \phi_n\}$.
Optional Input vb : the volume bound solution. **min**: solution minimality guarantee σ
Output a counterexample σ , UNSAT or B-UNSAT _{vb} .

```

1:  $F_\downarrow \leftarrow \emptyset, D_\downarrow \leftarrow \emptyset$ 
2: while  $\top$  do
3:    $\phi \leftarrow \bigwedge(F_\downarrow)$ 
4:    $\phi_g \leftarrow G(\phi, D_\downarrow)$ 
5:    $\phi_g^\perp \leftarrow \phi_g \wedge Inc(\phi_g, D_\downarrow)$ 
6:   if SOLVE( $\phi_g$ ) = UNSAT then
7:     return UNSAT
8:   else
9:      $\sigma \leftarrow SOLVE(\phi_g^\perp)$ 
10:  if  $\sigma = \text{UNSAT}$  then
11:     $\sigma_m \leftarrow \text{MINIMIZE}(\phi_g, \text{min})$ 
12:     $D_\downarrow \leftarrow D_\downarrow \cup \{o \mid act \in \sigma_{min}\}$ 
13:    if  $vol(\sigma_{min}) > vb$  then
14:      if  $\neg \text{min}$  then  $\text{min} \leftarrow \top$ 
15:      else return B-UNSAT $vb$ 
16:    else
17:      if  $\sigma \models F$  then
18:        return  $\sigma$ 
19:      else  $F_\downarrow \leftarrow F_\downarrow \cup \{\phi_i \mid \sigma \not\models \phi_i\}$ 

```

Algorithm 2 G_A: ground a FOL⁺⁺ formula in a domain D_\downarrow .

Input a FOL⁺⁺ formula ϕ and a domain D_\downarrow .
Output a grounded quantifier-free formula ϕ_g over relational objects.

```

1: if MATCH( $\phi, \exists o : r \cdot \phi'$ ) then
2:    $o' \leftarrow \text{NEWACT}(r)$ 
3:    $\phi'_g \leftarrow o'.ext \wedge G\_A(\phi'[o \leftarrow o'], D_\downarrow)$ 
4:   return  $\phi_g$ 
5: if MATCH( $\phi, \forall o : r \cdot \phi'$ ) then
6:   return  $\bigwedge_{o': r \in D_\downarrow} o'.ext \Rightarrow G\_A(\phi'[o \leftarrow o'], D_\downarrow)$ 
7: if MATCH( $\phi, f(t_1, \dots, t_n)$ ) then
8:   return  $f(G\_A(t_1, D_\downarrow), \dots, G\_A(t_n, D_\downarrow))$ 
9: if MATCH( $\phi, A(r, p, val)$ ) then
10:   $i \leftarrow \text{NAT}()$ ;
11:  Compute GLB and LUB
12:   $F_\downarrow.add(bound_{agr}(\phi))$ ;
13:  return  $i$ 
14: return  $\phi$  ▷ The case if  $\phi$  is atomic.

```

4 Bounded Satisfiability Checking of FOL⁺⁺

In this section, we present our algorithm for FOL⁺⁺ bounded satisfiability checking. Sec. 4.1 introduces the incremental approximation-based algorithm TOOL, for bounded satisfiability checking of FOL⁺⁺. Sec. 4.2 presents the numerical approximations of the aggregated values to support summations, and Sec. 4.3 describes support for other aggregation operators.

4.1 FOL⁺⁺ Bounded Satisfiability Checking

A naive approach to FOL⁺⁺ bounded (by a bound vb) satisfiability checking is to first eagerly ground FOL⁺⁺ formulas according to the set of domains that contain vb relational objects and then solve the grounded formulas with SMT solvers. However, the naive approach does not scale due to the large size of the grounded formula and the vast number of combinations for composing the domain of vb relational objects from different classes.

In contrast, our solution TOOL offers an incremental approximation-based approach, inspired by LEGOS [22]. It begins with an under-approximated domain of relational objects (e.g., an empty domain) and uses it to create grounded formulas representing both over-approximations and under-approximations for the

input formula. TOOL detects unbounded unsatisfiability if the over-approximation is unsatisfiable and finds an (optionally minimal) solution if the under-approximation has a satisfying solution. Otherwise, TOOL identifies a minimal cause of the unsatisfiability for the under-approximation and uses it to refine the current domain of relational objects. Stopping conditions are checked to detect bounded unsatisfiability during the domain expansion. A new iteration of approximation and refinement is performed on the refined domain. Eventually, TOOL derives the satisfiability result. In the following, we describe the detailed workflow of TOOL.

FOL⁺⁺ Bounded Satisfiability Checking with TOOL. Given a set of FOL⁺⁺ formulas $F = \{\phi_1, \dots, \phi_n\}$ and a bound $vb \in \mathbb{N}$, TOOL (Alg. 1) searches for a satisfying solution σ to F (i.e., $\sigma \models \phi_1 \wedge \phi_2 \dots \phi_n$) with a volume bounded by vb . TOOL either returns σ , UNSAT if ϕ is unsatisfiable, or B-UNSAT_{vb} (bounded UNSAT) if there is no satisfying solution within the bound vb . TOOL can be configured to find σ with a minimal volume if it is called with the option `-min`.

TOOL follows the incremental approximation and refinement methodology proposed by LEGOS [22]. The algorithm initially creates an under-approximated domain of relational objects D_\downarrow and the under-approximated formula F_\downarrow (L: 1). Then, it uses the algorithm G_A (Alg. 2) to ground the formula $\bigwedge(F_\downarrow)$, denoted as ϕ , in D_\downarrow , creating a quantifier-free FOL formula ϕ_g that over-approximates the satisfiability of ϕ .

Algorithm G_A grounds ϕ with boolean and numerical over-approximations. For boolean over-approximations, G_A grounds existential quantified expressions by instantiating the quantified relational objects with new relational objects. The resulting grounded expression is a boolean over-approximation because the newly instantiated relational objects are not in D_\downarrow , and hence are not constrained by universally quantified expressions in ϕ (L: 5 of Alg. 2). To handle aggregations in ϕ , G_A encodes their numerical over-approximations together with constraints on their global lower-bound (GLB) and local upper-bound (LUB) (L: 9-12 of Alg. 2). Numerical over-approximations, GLB and LUB are described in Sec. 4.2.

TOOL also creates an under-approximation ϕ_g^\perp (L: 5) by constraining every newly instantiated relational object o in ϕ_g to be semantically equivalent to some existing relational object o' in the current domain D_\downarrow (denoted as $Inc(\phi_g, D_\downarrow)$), where the semantically equivalent relation between relational objects o and o' (i.e., $o \equiv o'$) is defined as $CLS(o) = CLS(o')$ and $\bigwedge_{i=0}^{\iota(CLS(o))} (o[i] = o'[i]) \wedge o.ext = o'.ext \wedge o.time = o'.time$. With the constraints $Inc(\phi_g, D_\downarrow)$, both the propositional and numerical over-approximations in ϕ_g collapse to their semantic definitions in domain D_\downarrow .

The approximations, ϕ_g and ϕ_g^\perp , are solved by an SMT solver. If ϕ_g is unsatisfiable, then the original formula ϕ is also unsatisfiable (L: 7). If ϕ_g^\perp has a solution, then the solution is also valid for F_\downarrow (L: 18). TOOL then checks the solution σ against F (L: 17) and refines F_\downarrow by including formulas ϕ_i that are falsified by σ (L: 19). In case ϕ_g is satisfiable while ϕ_g^\perp is not, the algorithm expands the domain D_\downarrow to eliminate the 'smallest' cause of unsatisfiability by adding the new relational objects in the minimum (with respect to volume) so-

lution to ϕ_g . These relational objects are then included in D_\downarrow to eliminate the cause of unsatisfiability of ϕ_g^\perp in the future. Alternatively, if the minimal solution guarantee is not required, **TOOL** instead computes a solution that requires the minimal extension to D_\downarrow (which might have a higher volume than the minimal solution). With D_\downarrow expanded, **TOOL** starts a new round of approximation and expansion. However, during the domain expansion, if the volume of the minimal solution σ_m to the over-approximation is greater than the bound vb , the volume of any satisfying solution to ϕ would also be greater than vb . In such cases, **TOOL** returns **B-UNSAT** _{vb} .

The major difference between **TOOL** and **LEGOS** is the former’s ability to effectively handle aggregation with numerical over and under-approximation (L: 4- and 5) discussed in Sec. 4.2 and 4.3. Other differences include incremental solving and relaxed domain search discussed in Appendices G and H, respectively.

Example. Consider the example, without aggregation, in Sec. 2. Let $\phi := \forall a : A. (\exists a', a'' : A \cdot (a.val = a'.val + a''.val) \wedge \exists a : A \cdot a.val = 5)$ and $vb = 5$. Initially, $D_\downarrow = \emptyset$ and $\phi_g = a_1.val = 5$, where a_1 is instantiated by G_A . The over-approximation ϕ_g is satisfiable, and the under-approximation ϕ_g^\perp is **UNSAT**. Suppose σ_{min} contains a_1 , and thus D_\downarrow is expanded with a_1 . In the second iteration, ϕ_g introduces a new constraint $a_1.val = a_2.val + a_3.val$, where a_2 and a_3 are instantiated by G . The formula ϕ_g is satisfiable and ϕ_g^\perp is **UNSAT**. Suppose σ_{min} contains a_1 and a_2 , and a_2 is added to D_\downarrow . In the third iteration, ϕ_g introduces a constraint $a_2.val = a_4.val + a_5.val$. The under-approximation ϕ_g^\perp can be satisfied in $D_\downarrow = a_1, a_2$ with the solution σ shown in Sec. 2. However, if $vb = 1$, then **TOOL** returns **B-UNSAT**₁ in the second iteration since $|D_\downarrow| = |a_1, a_2| > 1$.

4.2 Supporting Aggregation

Recall from Sec. 4.1 that **TOOL** (Alg. 1) checks the satisfiability of an FOL^{*+} formula ϕ using its over- and under-approximations, computed on L: 4 (by calling Alg. 2 and L: 5, respectively). Suppose ϕ contains constraints on aggregated values. We need to ensure that G_A correctly encodes aggregated values in the over- and under-approximations of ϕ . More specifically, **TOOL** needs to preserve two invariants: (1) if the over-approximation ϕ_g is **UNSAT**, then the formula is **UNSAT**; (2) every solution to the under-approximation ϕ_g^\perp is a solution to ϕ . Note that the over- and under-approximations of the aggregated value are not the conventional upper and lower bounds. Instead, they approximate with respect to the satisfiability of ϕ . An aggregated value could be over-approximated with a lower value than its under-approximated value to make ϕ more satisfiable (e.g., if $\phi = \text{Sum}(S, p, val) < 10$, then decreasing the value of $\text{Sum}(S, p, val)$ makes ϕ more satisfiable). In this subsection, we focus on the support for *Sum*. Support for the other aggregation functions is described in Sec. 4.3.

Under-approximation. Suppose $\text{Sum}(S, p, val)$ is a summation used in formula ϕ , and D_\downarrow is an under-approximated domain of relational objects (i.e., $D_\downarrow \subseteq D$ for some D that has a solution to ϕ). We define the under-approximated sum in D_\downarrow as $\text{Sum}_{D_\downarrow}(S, p, val) := \sum_{s \in S_{D_\downarrow}} \text{ite}(s.\text{ext} \wedge p(s), val(s), 0)$. The value

$$\begin{aligned}
\text{GLB}(t, D_\downarrow) &= \begin{cases} t & \text{if } t = v \mid c \\ -\text{LUB}(t_1, D_\downarrow) & \text{if } t = -t_1 \\ \text{GLB}(t_1, D_\downarrow) + \text{GLB}(t_2, D_\downarrow) & \text{if } t = t_1 + t_2 \\ c \times \text{ite}(c \geq 0, \text{GLB}(t_1, D_\downarrow), \text{LUB}(t_1, D_\downarrow)) & \text{if } t = c \times t_1 \\ \sum_{s \in S_{D_\downarrow}} \text{ite}(\neg s.\text{ext} \vee \text{G_A}(\neg p(s), D_\downarrow), 0, \text{GLB}(\text{val}(s), D_\downarrow)) & \text{if } t = \text{Sum}(S, p, \text{val}) \end{cases} \\
\text{LUB}(s, D_\downarrow) &= \begin{cases} t & \text{if } t = v \mid c \\ -\text{GLB}(t_1, D_\downarrow) & \text{if } t = -t_1 \\ \text{LUB}(t_1, D_\downarrow) + \text{LUB}(t_2, D_\downarrow) & \text{if } t = t_1 + t_2 \\ c \times \text{ite}(c \geq 0, \text{LUB}(t_1, D_\downarrow), \text{GLB}(t_1, D_\downarrow)) & \text{if } t = c \times t_1 \\ \sum_{s \in S_{D_\downarrow}} \text{ite}(s.\text{ext} \wedge \text{G_A}(p(s), D_\downarrow), \text{LUB}(\text{val}(s), D_\downarrow), 0) & \text{if } t = \text{Sum}(S, p, \text{val}) \end{cases}
\end{aligned}$$

Fig. 2. The functional definition of GLB and LUB for Def. 4 and Def. 5.

of $\text{Sum}_{D_\downarrow}(S, p, \text{val})$ is an under-approximation because S is restricted by D_\downarrow , and D_\downarrow is an under-approximation of D . For brevity, given a summation sum and a domain D , we refer to the under-approximation of sum in D as sum_D .

Over-approximation. Suppose $\text{Sum}(S, p, \text{val})$ is a summation and $\text{Sum}_{D_\downarrow}(S, p, \text{val})$ is its under-approximation D_\downarrow . The value of $\text{Sum}(S, p, \text{val})$ can be over-approximated in three different ways: (1) decreasing its value by making relational objects in S falsify p and decrease the output of val ; (2) increasing its value by making relational objects in S satisfy p and increase the output of val ; or (3) increasing its value by extending D_\downarrow to include more relational objects in S . Methods (1) and (2) are *local approximations*, as they might not need to expand D_\downarrow . However, the values of the local approximation are bounded by the global lower-bound (GLB) and the local upper bound (LUB) of $\text{Sum}(S, p, \text{val})$.

Definition 2 (Global lower-bound). Let $\text{sum} = \text{Sum}(S, p, \text{val})$ be a summation, and D_\downarrow be a domain. $\text{GLB}_{D_\downarrow}^{\text{sum}}$ is a global lower bound of sum in D_\downarrow if and only if for every domain D_{D_\downarrow} , $\text{GLB}_{D_\downarrow}^{\text{sum}} \leq \text{sum}_D$, where $\text{sum}_D = \sum_{s \in S_D} \text{ite}(s.\text{ext} \wedge p(s), \text{val}(s), 0)$ is the under-approximated summation in D .

Definition 3 (Local upper-bound). Let sum be a summation in the form of $\text{Sum}(S, p, \text{val})$, and D_\downarrow be a domain. $\text{LUB}_{D_\downarrow}^{\text{sum}}$ is a local upper-bound of sum in domain D_\downarrow if and only if $\text{LUB}_{D_\downarrow}^{\text{sum}} \geq \text{sum}_{D_\downarrow}$, where $\text{sum}_{D_\downarrow} = \sum_{s \in S_{D_\downarrow}} \text{ite}(s.\text{ext} \wedge p(s), \text{val}(s), 0)$ is the under-approximated summation in D_\downarrow .

Given a summation and a domain, we can compute the global lower-bound and local upper-bound using the functions GLB and LUB, respectively.

Definition 4 (Global lower-bound function). GLB is a function that receives a numerical term t and a domain D_\downarrow and computes $\text{GLB}(t, D_\downarrow)$ as shown in Fig. 2, where G_A is the extended version of G (Alg. 2) for computing the over-approximation of an FOL^{*+} formula, and LUB is a function that computes a numerical term's local upper bound (Def. 5).

Definition 5 (Local upper-bound function). LUB is a function that receives a numerical term t and a domain D_\downarrow and computes $\text{LUB}(t, D_\downarrow)$ as shown in Fig. 2. GLB is a function that computes a term's global lower bound (Def. 4).

To summarize, given an under-approximated summation $Sum_{D_\downarrow}(S, p, val) = \sum_{s \in S} ite(p(s), val(s), 0)$, we can obtain its LUB by over-approximating $p(s)$ as $GA(p(s), D_\downarrow)$ and over-approximating $val(s)$ as $LUB(val(s), D_\downarrow)$. We can obtain the GLB by over-approximating $\neg p(s)$ (and swapping the expressions in ite 's “then” and “else” branches), and under-approximating $val(s)$ as $GLB(val(s), D_\downarrow)$.

Lemma 1 (Correctness of GLB and LUB). *For every domain D_\downarrow and sum in the form of $Sum(S, p, val)$, $GLB(sum, D_\downarrow)$ computes a global lower-bound, and $LUB(sum, D_\downarrow)$ computes a local upper-bound.*

The proof of Lemma 1 is given in Appendix A.

Corollary 1. *Let sum be a summation in the form of $Sum(S, p, val)$. If for every $s \in S$, $p(s)$ and $val(s)$ do not contain summations or quantifiers, then $GLB(sum, D) = LUB(sum, D) = Sum_D(S, p, val)$ for every domain D .*

Given a summation sum in the form $Sum(S, p, val)$ and a domain D_\downarrow , GLB computes the lower-bound of summations for every domain which extends D_\downarrow . Since the extension is monotone, $GLB(sum, D_\downarrow)$ is always a valid lower-bound (i.e., global), and hence is a lower-bound for sum . On the other hand, if the over-approximation of sum exceeds $LUB(sum, D_\downarrow)$, then the approximation is not local, and it must extend D_\downarrow to include more relational objects in S . Therefore, the non-local approximation creates an obligation to expand D_\downarrow .

Definition 6 (Over-approximation). *Given a sum in the form of $Sum(S, p, val)$ and a domain D_\downarrow , let $GLB_{D_\downarrow}^{sum}$ and $LUB_{D_\downarrow}^{sum}$ be the sum's global lower-bound and local upper-bound at D_\downarrow , respectively. The over-approximation of sum is a new integer variable i that satisfies the following constraints, denoted as $bound_{agr}(sum)$:*

- (a) $i \geq GLB_{D_\downarrow}^{sum}$
- (b) if $i > LUB_{D_\downarrow}^{sum}$ then $\exists s' \cdot p(s') \wedge val(s') + Sum(S, \lambda s : s \neq s' \wedge p(s), val) = i$

Constraint (a) ensures that the over-approximation of sum respects the global lower-bound, and constraint (b) handles the case of non-local approximation where D_\downarrow needs to be expanded. More specifically, if the over-approximation i is greater than $GLB_{D_\downarrow}^{sum}$, then D_\downarrow should be extended with a new relational object s' satisfying p . A new summation is introduced in rule (b) to be equal to the remaining value of the summation over the set $S \setminus s'$. Conversely, if the domain expansion is not possible, then sum is upper-bounded by $LUB_{D_\downarrow}^{sum}$.

Example: Consider the property P_1 stated in English in Fig. 1(a). It contains the condition $C1$ that compares the size of user transactions with their daily spending of the last 7 days. The daily transaction amount made by a user u at day t is expressed as a summation sum in the form $Sum(Trans, \lambda tr' : tr'.u = u \wedge tr'.time = t, \lambda tr' : tr'.x)$. Suppose the domain D_\downarrow consists of three relational objects $\{tr_1, tr_2, tr_3\}$ of class $Trans$. The under-approximation sum_D is $ite(tr_1.u = u \wedge tr_1.time = t, tr_1.x, 0) + ite(tr_2.u = u \wedge tr_2.time = t, tr_2.x, 0) + ite(tr_3.u = u \wedge tr_3.time = t, tr_3.x, 0)$. The over-approximation

of sum is a fresh integer variable i such that $i \geq \text{GLB}(sum, D_\downarrow)$ and $i \geq \text{LUB}(sum, D_\downarrow) \implies \exists tr : \text{Trans} \cdot i = tr.x + \text{Sum}(\text{Trans}, \lambda tr' : tr'.u = u \wedge tr'.time = t \wedge tr \not\equiv tr', \lambda tr' : tr'.x)$. Since sum is stratified at layer 0 and $p(s)$, $\text{GLB}(sum, D_\downarrow) = \text{LUB}(sum, D_\downarrow) = sum_{D_\downarrow}$ (Cor. 1). An example summation stratified at a higher level is given in Appendix C.

Over-approximation & under-approximation for FOL⁺⁺. We now explain how to encode the over- and under-approximations for FOL⁺⁺ formulas by incorporating the over- and under-approximations of summations. Given an FOL⁺⁺ formula ϕ and a domain D_\downarrow , the over-approximation of ϕ , denoted as ϕ_g , is a quantifier-free formula computed by $\text{G_A}(\phi, D_\downarrow)$, where every summation in ϕ is encoded (L: 9 in Alg. 2) according to the over-approximation definition (Def. 6). The under-approximation of ϕ in D_\downarrow , denoted as ϕ_g^\perp , is the formula $\phi_g \wedge \text{Inc}(\phi_g, D_\downarrow)$ (L: 5 of Alg. 1), where $\text{Inc}(\phi_g, D_\downarrow)$ requires every newly instantiated relational object o in ϕ_g to be semantically equivalent to some existing relational object o' in the current domain D_\downarrow . Note that under the constraints $\text{Inc}(\phi_g, D_\downarrow)$, both the GLB and LUB are collapsed to the under-approximated sum sum_{D_\downarrow} , resulting in $sum = sum_{D_\downarrow}$ in ϕ_g^\perp .

Theorem 1 (Over-approximation soundness). *Given an FOL⁺⁺ formula ϕ and a domain D_\downarrow , $\text{G_A}(\phi, D_\downarrow)$ is an over-approximation of ϕ (i.e., if there exists a domain D where ϕ is satisfiable, then $\text{G_A}(\phi, D_\downarrow)$ is also satisfiable).*

Theorem 2 (Under-approximation soundness). *Given an FOL⁺⁺ formula ϕ and a domain D_\downarrow , let ϕ_g be the over-approximation computed by $\text{G_A}(\phi, D_\downarrow)$, and let $\phi_g^\perp = \phi_g \wedge \text{Inc}(\phi_g, D_\downarrow)$ where every sum in ϕ_g is under-approximated by sum_{D_\downarrow} . Then ϕ_g^\perp is an under-approximation of ϕ (i.e., if ϕ_g^\perp has a solution, then it must be a solution to ϕ).*

Proofs of Thm. 1 and Thm. 2 are provided in Appendix A and B, respectively.

To efficiently manage summations, each summation sum is treated as an auxiliary relational object. If the relational object for sum is not within the domain D_\downarrow , it is treated as a fresh variable i . Otherwise, the constraints in $bound_{agr}(sum)$ are applied during the process denoted as G_A . If $i > \text{LUB}(sum, D_\downarrow)$, the second constraint of $bound_{agr}(sum)$ expands D_\downarrow by introducing a relational object s' and a new summation. This new summation can then be over-approximated within the expanded domain, potentially triggering subsequent rounds of domain expansion. We illustrate TOOL on an FOL⁺⁺ formula with aggregation in Appendix D.

4.3 Support for Other Aggregation Functions

We have been focusing on the support for the aggregation function Sum . Other aggregation functions in FOL⁺⁺, Count , Max , and Min , are supported analogously, over and under-approximation. More specifically, the under-approximation of aggregation is bounded by the current domain, while its over-approximation is bounded by its global lower bound (GLB) and local upper bound (LUB). In

this section, we present the under-approximation, GLB, and LUB for *Max*. We also demonstrate that the support for *Count* and *Min* can be realized via the support for *Sum* and *Max*, respectively.

Count. *Count* has the signature $Count(S, p)$, where S is a class and p is a predicate. It is equivalent to $Sum(S, p, One())$, where $One()$ is a constant function returning one. Thus, the support for *Count* is realized through that of *Sum*.

Max. *Max* has the signature $Max(S, p, val)$, where S is a class, p is a predicate, and val is a numerical function. We support *Max* using over- and under-approximation. In a domain D , the under-approximation is $\mathbf{GV}(\{ite(s.ext \wedge p(s), val(s), -\infty) \mid s \in S_D\})$ where $\mathbf{GV}(a, b)$ returns the greater value of a and b . The over-approximation in a domain D is a fresh integer variable i under the constraint $bound_{agr}(max)$: (1) i must be no less than its global lower-bound (GLB_D^{max}), and (2) if i is greater than its local upper-bound (LUB_D^{max}), then there exists a relational object s of class S such that $p(s) \wedge val(s) = i$ where

$$\begin{aligned} GLB_D^{max} &= \max(\{ite(\neg s.ext \vee G_A(\neg p(s), D), -\infty, GLB(val(s), D)) \mid s \in S_D\}) \\ LUB_D^{max} &= \max(\{ite(s.ext \wedge G_A(p(s), D), GLB(val(s), D), -\infty,) \mid s \in S_D\}). \end{aligned}$$

Intuitively, we can obtain the LUB of $Max(S, p, val)$ by over-approximating $p(s)$ as $G_A(p(s), D_\downarrow)$ and over-approximating $val(s)$ as $LUB(val(s), D)$. We can obtain the GLB by over-approximating $\neg p(s)$ (and swapping in *ite* its “then” and “else” branches), and under-approximating $val(s)$ as $GLB(val(s), D)$. Note that the shapes of the GLB, LUB, and over and under-approximations for *Max* are identical to the ones for *Sum*. The only difference between them are the value aggregation functions ($\sum \rightarrow max$) and the default value ($0 \rightarrow -\infty$).

Min. *Min* has the signature $Min(S, p, val)$, and it is equivalent to $-Max(S, p, -val)$. Therefore, the support for *Min* is realized via the support for *Max*.

To summarize, we proposed a general method to support aggregation in FOL^{*+} with over- and under-approximations. The ‘secrete sauce’ is the numerical approximation functions (LUB and GLB) that over/under-approximate the values of numerical terms in FOL^{*+} . One can use our method to support new aggregation functions by defining their LUB and GLB.

5 Evaluation

In this section, we report on experiments aimed at studying the effectiveness of **TOOL** in supporting bounded satisfiability checking of FOL^{*+} . Specifically, we compare **TOOL** with **LEGOS** (the baseline) on FOL^{*+} benchmarks to answer two questions: **RQ1**: How does the aggregation support of **TOOL** compare in effectiveness to **LEGOS** when dealing with instances that have constraints on aggregated values? **RQ2**: How does the aggregation support of **TOOL** impact the performance cost compared to **LEGOS** for instances that don’t have constraints on aggregated values? The experiments were conducted using several case studies from the literature on a Silicon Mechanics Rackform iServ R331.v4 with 12-core

Intel E5-2697v2 CPUs, running 64-bit Ubuntu GNU/Linux 8. The runtime and memory of each instance were limited to 5000 seconds and 32GB, respectively. Correctness of TOOL was verified by ensuring output consistency with LEGOS and manual inspection when LEGOS timed-out.

Case studies. To answer RQ1, we collected five FOL*⁺ case studies that contain formulas with constraints on the aggregated data values, denoted as *Instances-A*. These correspond to a banking system handling client transactions (BKG) [18], rules for constructing and maintaining a connected graph over time (DGraph) [17], synthesizing the function $f(x) = \text{Count}(y)f(y) = x$ on a finite domain (Magic) [16], a social activity system that manages player matching over time (Player), and system for keeping track of personal health formulas previously using a walk-around [5, 20] rewritten using aggregation (PHIM-A). Since LEGOS does not support aggregation directly, aggregation in these benchmarks is modelled in FOL* similarly to the ones shown in Fig. 1 following the recursive definition of aggregation [35]. The modelling methodology is in Appendix F.

To answer RQ2, we use five FOL* case studies (converted from MFOTL) from LEGOS [22]. These do not contain aggregation and are denoted as *Instances-NA*: a banking system that processes customer transactions (BST) [11], a system for monitoring COVID patients at home and enabling doctors to monitor patient data (CF@H) [32], an approval policy for publishing business reports (PBC) [11], an air-traffic control system design that aims to avoid aircraft collisions (NASA) [24, 34], and a system for keeping track of personal health (PHIM) [5, 20].

These case studies cover a broad spectrum of characteristics, ranging from no quantified data constraints (NASA) to a large number of data constraints (CF@M, PHIM, BKG). They also vary in their usage of the aggregation operator, from not using it (PHIM) to using it (Magic, BKG, DGraph, Player, and PHIM-A), and to using it heavily (BKG). Two of these are real-world case studies (CF@H and NASA) and five are published regulatory compliance checking examples (PBC, BST, PHIM, PHIM-A, and BKG).

RQ1. To answer RQ1, we evaluate TOOL and LEGOS on *Instances-A* where aggregation is modelled in FOL* for LEGOS. We report the outcome and runtime for both tools with (min) and without (n-min) minimal solution guarantee. i.e., 52 trials on 26 instances for each tool. The results are shown in Tab. 1. We observed that supporting aggregation directly in TOOL yielded a significantly better performance than modelling aggregation in FOL*. TOOL succeeded on all 26 instances and 50 out of 52 trials, whereas LEGOS succeeded on only 9 out of 26 instances and 15 out of 52 trials. TOOL is on average 19 times faster (geometric mean) than LEGOS on instances where some configurations of both TOOL and LEGOS succeed. LEGOS’s runtime scaled much worse than TOOL as the difficulty of instances increased for case studies Player, PHIM-A, and Magic Sequences. For case studies Graph and BKG, where aggregated values are heavily constrained, LEGOS timed out even on the simplest instances.

We also noticed a significant performance cost of the minimal solution guarantee for both TOOL and LEGOS, which aligns with the findings in [22]. Running TOOL without the minimal solution guarantee solved two additional instances

and was 3.6 times more efficient (geometric mean) than TOOL-min. Additionally, TOOL-min successfully improved the volume of the solution for 7 out of 20 satisfiable instances that were solved by both configurations.

We answer RQ1 positively: TOOL’s aggregation support significantly improves the effectiveness of FOL⁺⁺ bounded satisfiability checking.

RQ2. To answer RQ2, we compared TOOL and LEGOS on the benchmark *Instances-NA*. We report the run-time for both tools with (min) and without (n-min) the minimal solution guarantee (4 configurations). The results are shown in Tab. 2. We observed that performance of TOOL is competitive with respect to LEGOS on *Instances-NA*. All instances and all trials were successfully analyzed by both TOOL and LEGOS with a total run-time of 83.28 seconds and 475.65 seconds (5.7x total time improvement), respectively. The geometric mean time of TOOL compared to LEGOS is 88.9% (81% and 96% with and without the minimal solution guarantee, respectively).

TOOL performed significantly better for hard instances where LEGOS (either min or n-min) took more than 5 seconds (highlighted in Tab. 2). The average run-time for TOOL and LEGOS was 6.51 seconds and 45.78 seconds (7x faster), respectively. The geometric mean time of TOOL compared to LEGOS was 63.7% (80% and 33% with and without minimal solution guarantee, respectively). On the other hand, for the easy instances where LEGOS finished within 5 seconds, TOOL was competitive with LEGOS: the average run-time for TOOL and LEGOS was 0.36 seconds and 0.355 seconds, respectively, and the geometric mean time of TOOL compared to LEGOS was 105%. The performance differences on easy instances were small but the benefit of TOOL on hard instances significantly outweighed the cost of the overhead.

We answer RQ2 positively: TOOL is competitive with LEGOS for bounded satisfiability checking on instances without aggregation, and it performs significantly better on harder instances.

6 Related Work

Reasoning with Aggregation. Aggregation has been extensively studied in logic programming and database theory and is supported by reasoning tools. Aggregation is commonly supported in Datalog [33] by Datalog engines such as Soufflé [26], Socialite [40, 41] and BigDatalog [42]. The more expressive logic system answer set programming (ASP) [8, 19, 23] has also developed support for aggregation in solvers such as DLV [1] and WASP [2]. In Datalog and ASP, aggregation semantics is defined over sets of grounded relations with a fixed domain of atoms (close-world assumption). In contrast, TOOL supports aggregation without requiring a fixed domain of atoms. TOOL can include new relational objects to a set as long as they do not conflict with the requirements. Additionally, Datalog and ASP operate on a restricted form of FOL whereas we support aggregation on a more general basis that allows arbitrarily quantifier alternation over relational objects. The Declarative Modeling Language Alloy [25] and Electrum [14] support aggregation and quantifiers over sets of objects instantiated

Table 1. Comparison between TOOL and LEGOS for bounded satisfiability checking on FOL⁺⁺ case studies (with aggregations). The satisfiability result and time (res | time(s)) are reported for four configurations: TOOL with (min) and without (n-min) minimal solution guarantee and LEGOS with (min) and without (n-min) minimal solution guarantee. The satisfiability results are one of the following: the volume of a solution (e.g., 5), unsatisfiable (U), bounded UNSAT (BU), or Timeout (T/O): runtime exceeds 5000 sec. Instances from the same case study are roughly ordered by their difficulties (e.g, *pl2* is a more complicated more *pl1*).

Case Studies		TOOL		LEGOS		Case Stuides		TOOL		LEGOS	
		min	n-min	min	n-min			min	n-min	min	n-min
DGraph	dg1	8 0.96s	8 0.66s	TO	TO	Magic Seq	m2	U 0.2s	U 0.2s	U, 1.6s	U, 1.3s
	dg2	12 2.45s	12 1.59s	TO	T/O		m4	2 0.12s	2 0.08s	2, 1.77s	14, 0.39s
	dg3	12 11.4s	12 1.61s	T/O	T/O		m10	4 1.51s	4 1.13s	T/O	14, 1092s
	dg4	16 10.4s	16 4.09s	T/O	T/O		m1H	4 1.67s	4 1.19s	T/O	13, 9.16s
	dg5	20 70.2s	20 14.3s	T/O	T/O		m1K	4 1.95s	4 1.37s	T/O	13, 8.94s
Player	pl1	2 0.08s	2 0.05s	2 0.5s	2 0.47s	BKG (Day)	bd1	U 0.77s	U 1.02s	T/O	T/O
	pl2	5 40.64s	9 1.54s	T/O	T/O		bd2	7 19.1s	9 1.20s	T/O	T/O
	pl3	T/O	12 3.31s	T/O	T/O		bd3	12 821.2s	13 23.99s	T/O	T/O
	pl4	T/O	16 53.13s	T/O	T/O		bd4	20 1917s	21 333.9s	T/O	T/O
PHIM-A	ph7	19 2.35s	20 1.66s	19 14.06s	22 8.36s	BKG (Hour)	bd5	U 230.7s	U 204.8s	T/O	T/O
	ph8	22 4.2s	23 3.18s	22 204.4s	27 204.9s		bh1	10 87.6s	12 12.95s	T/O	T/O
	ph9	25 8.49s	25 7.37s	25 400.1s	32 383.6s		bh2	15 70.8s	15 26.60s	T/O	T/O
	ph10	32 296s	32 294.0s	T/O	T/O		bh3	U 5.00s	U 3.92s	T/O	T/O

with a bounded data domain. TOOL differs from them as it (optionally) bounds the number of relational objects instead of the data domain. MONPOLY [12] enables runtime monitoring of policies specified in Metric First-Order Logic with Aggregations [11] (MFOTL_ω). This is the formalism most closely aligned with FOL⁺⁺ in terms of expressiveness. Complementary to runtime monitoring, TOOL provides satisfiability checking for FOL⁺⁺, which captures the monitorable fragment of MFOTL_ω.

Bounded Satisfiability Checking. Satisfiability checking of temporal logic has been studied to verify system designs. Approaches targeting various temporal logic, such as linear temporal logic (LTL) [13, 27–29, 31, 39], metric temporal logic (MTL) [38], mission-time LTL [30] and signal temporal logic (STL) [6] have been proposed. Satisfiability checking for first-order logic is supported by SMT solvers such as Z3 [36] and CVC5 [7]. To verify LP in early system designs, LEGOS [22] was developed to support bounded satisfiability checking for MFOTL, which extends MTL with first-order quantifiers to capture data constraints. LEGOS translates MFOTL formulas into FOL*, and then checks the satisfiability of the translated formula. In contrast, TOOL checks the bounded satisfiability of FOL⁺⁺, which is a super-set of FOL* (with aggregations extension). TOOL supports reasoning on aggregations with numerical over and under-approximation.

Table 2. Run-time performance of TOOL and LEGOS on benchmark *Instances-NA* (without aggregations). The satisfiability result and running time (time, in seconds) are reported for four configurations: TOOL with (min) and without (n-min) minimal solution guarantee and LEGOS with (min) and without (n-min) minimal solution guarantee. Significant performance differences are highlighted: red and green indicate shorter and longer execution time, respectively

Case studies		Result	TOOL		LEGOS	
			min	n-min	min	n-min
BST	bst1	UNSAT	0.29	0.3	0.34	0.35
	bst2	SAT	0.11	0.04	0.04	0.04
	bst3	SAT	0.45	0.29	0.92	0.52
NASA	na1	UNAST	0.66	0.67	0.87	0.89
	na2	UNAST	0.18	0.18	0.2	0.2
	na3	UNAST	0.19	0.19	0.2	0.2
	na4	UNAST	0.67	0.68	0.89	0.84
	na5	UNAST	0.11	0.11	0.11	0.11
	na6	UNAST	0.02	0.03	0.03	0.03
	na1b	UNAST	0.14	0.14	0.14	0.14
	na2b	UNAST	0.14	0.14	0.14	0.14
	na3b	UNAST	0.15	0.14	0.15	0.15
	na4b	UNAST	0.16	0.16	0.14	0.15
	na5b	UNAST	0.14	0.14	0.14	0.14
	na6b	UNAST	0.03	0.03	0.03	0.03

Case Studies		Result	TOOL		LEGOS	
			min	n-min	min	n-min
CF@H	cf1	UNSAT	1.23	1.59	1.61	1.78
	cf2	SAT	3.15	2.37	10.34	2.4
	cf3	SAT	5.09	3.82	5.98	3.14
	cf4	SAT	4.47	3.44	90.28	4.42
	cf5	SAT	1.26	0.95	0.5	0.47
	cf6	SAT	1.28	0.98	0.49	0.47
	cf7	UNSAT	0.61	0.61	0.31	0.32
PBC	pbc	SAT	0.51	0.48	0.36	0.39
PHIM	ph1	UNSAT	0.05	0.05	0.05	0.04
	ph2	UNSAT	0.04	0.04	0.03	0.03
	ph3	UNSAT	0.06	0.06	0.05	0.05
	ph4	UNSAT	0.05	0.05	0.05	0.05
	ph5	SAT	14.4	10.72	25.68	15.67
	ph6	UNAST	0.78	0.79	1.24	1.23
	ph7	SAT	10.97	6.7	268.11	31.84

7 Conclusion

In this paper, we extended FOL* with aggregation, calling the resulting language FOL⁺⁺. We also developed TOOL, a bounded satisfiability checking tool-supported approach for FOL⁺⁺. TOOL enables a direct specification and an efficient reasoning over constraints on aggregated values. TOOL supports aggregation by over- and under-approximating aggregated values and incrementally refining the approximations. Compared to the state-of-the-art bounded satisfiability checker for FOL*, LEGOS, TOOL is up to 10 times more efficient on instances with constraints on aggregated values. Moreover, it performs as effectively, if not better, on other instances.

In the future, we aim to improve TOOL’s efficiency for computing the minimal solution, especially when aggregation constraints are present. This could be achieved by integrating existing work from Datalog [42], ASP [23], and SMT solvers [16]. Finally, we would like to move beyond bounded satisfiability to discover and capture solutions with an infinite volume.

References

1. Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV2. In: Balduccini, M., Janhunen, T. (eds.) Logic Programming and Nonmonotonic Reasoning - 14th In-

- ternational Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10377, pp. 215–221. Springer (2017). https://doi.org/10.1007/978-3-319-61660-5_19
2. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: Cabalar, P., Son, T.C. (eds.) Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2013), Corunna, Spain. LNCS, vol. 8148, pp. 54–66. Springer (2013). https://doi.org/10.1007/978-3-642-40564-8_6
 3. Alviano, M., Greco, G., Leone, N.: Dynamic Magic Sets for Programs with Monotone Recursive Aggregates. In: Delgrande, J.P., Faber, W. (eds.) Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6645, pp. 148–160. Springer (2011). https://doi.org/10.1007/978-3-642-20895-9_14
 4. Anonymous: Repository for TACAS Submission : Bounded Satisfiability Checking of FOL* Formulas with Aggregations (2023), <https://anonymous.4open.science/r/TACASTOOL-491C/>
 5. Arfelt, E., Basin, D.A., Debois, S.: Monitoring the GDPR. In: Sako, K., Schneider, S.A., Ryan, P.Y.A. (eds.) Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11735, pp. 681–699. Springer (2019). https://doi.org/10.1007/978-3-030-29959-0_33
 6. Bae, K., Lee, J.: Bounded model checking of signal temporal logic properties using syntactic separation. Proc. ACM Program. Lang. **3**(POPL), 51:1–51:30 (2019). <https://doi.org/10.1145/3290364>
 7. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2022), Munich, Germany. LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
 8. Bartholomew, M., Lee, J., Meng, Y.: First-Order Semantics of Aggregates in Answer Set Programming Via Modified Circumscription. In: Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21-23, 2011. AAAI (2011)
 9. Basin, D.A., Klaedtke, F., Marinovic, S., Zalinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods Syst. Des. **46**(3), 262–285 (2015). <https://doi.org/10.1007/s10703-015-0222-7>
 10. Basin, D.A., Klaedtke, F., Müller, S.: Policy monitoring in first-order temporal logic. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 1–18. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_1, https://doi.org/10.1007/978-3-642-14295-6_1
 11. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). <https://doi.org/10.1145/2699444>, <https://doi.org/10.1145/2699444>
 12. Basin, D.A., Klaedtke, F., Zalinescu, E.: The monpoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime

- Verification Tools, September 15, 2017, Seattle, WA, USA. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017). <https://doi.org/10.29007/89hs>, <https://doi.org/10.29007/89hs>
13. Bersani, M.M., Frigeri, A., Morzenti, A., Pradella, M., Rossi, M., Pietro, P.S.: Constraint LTL satisfiability checking without automata. *J. Appl. Log.* **12**(4), 522–557 (2014). <https://doi.org/10.1016/j.jal.2014.07.005>
 14. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The electrum analyzer: model checking relational first-order temporal specifications. In: Huchard, M., Kästner, C., Fraser, G. (eds.) *Proceedings of the 33rd International Conference on Automated Software Engineering, (ASE 2018)*, Montpellier, France. pp. 884–887. ACM (2018). <https://doi.org/10.1145/3238147.3240475>
 15. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer (2008)
 16. Denecker, M., De Cat, B.: DPLL (Agg): An efficient SMT module for aggregates. In: *Logic and Search* (2010)
 17. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic Graph Algorithms. In: Atallah, M.J. (ed.) *Algorithms and Theory of Computation Handbook*. Chapman & Hall/CRC Applied Algorithms and Data Structures series, CRC Press (1999). <https://doi.org/10.1201/9781420049503-c9>
 18. EU payment services directive: Directive 2007/64/EC of the European Parliament and of the Council of 13 October 2000 establishing a framework for payment services in the internal market amending (2007), <https://eur-lex.europa.eu/eli/dir/2007/64/2009-12-07>
 19. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.* **175**(1), 278–298 (2011). <https://doi.org/10.1016/j.artint.2010.04.002>
 20. Feng, N., Marsso, L., Garavel, H.: Health record. Model checking context model (MCC’21), Dept. of Computer Science - University of Toronto (2021), <https://mcc.lip6.fr/pdf/HealthRecord-form.pdf>
 21. Feng, N., Marsso, L., Getir-Yaman, S., Beverley, T., Calinescu, R., Cavalcanti, A., Chechik, M.: Towards a Formal Framework for Normative Requirements Elicitation. In: *Proceedings of the 38th International Conference on Automated Software Engineering, (ASE’2023)*, Kirchberg, Luxembourg. IEEE (2023)
 22. Feng, N., Marsso, L., Sabetzadeh, M., Chechik, M.: Early Verification of Legal Compliance via Bounded Satisfiability Checking. In: *Proceedings of the 34th International Conference on Computer-Aided Verification (CAV’23)*, Paris, France. Lecture Notes in Computer Science, Springer (2023)
 23. Ferraris, P.: Logic programs with propositional connectives and aggregates. *ACM Trans. Comput. Log.* **12**(4), 25:1–25:40 (2011). <https://doi.org/10.1145/1970398.1970401>
 24. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: Automated air traffic control design space exploration. In: Chaudhuri, S., Farzan, A. (eds.) *Proceedings of the 28th International Conference on Computer Aided Verification (CAV’2016)*, Toronto, ON, Canada. LNCS, vol. 9780, pp. 3–22. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_1
 25. Jackson, D.: Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* **11**(2), 256–290 (2002). <https://doi.org/10.1145/505145.505149>
 26. Jordan, H., Scholz, B., Subotic, P.: Soufflé: On Synthesis of Program Analyzers. In: *Proceedings of the 28th International Conference on Computer Aided Verification*

- (CAV'2016), Toronto, ON, Canada. Lecture Notes in Computer Science, vol. 9780, pp. 422–430. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_23
27. Li, J., Pu, G., Zhang, L., Vardi, M.Y., He, J.: Accelerating LTL satisfiability checking by SAT solvers. *J. Log. Comput.* **28**(6), 1011–1030 (2018), <https://doi.org/10.1093/logcom/exy013>
 28. Li, J., Pu, G., Zhang, Y., Vardi, M.Y., Rozier, K.Y.: SAT-based explicit LTLf satisfiability checking. *Artif. Intell.* **289**, 103369 (2020). <https://doi.org/10.1016/j.artint.2020.103369>
 29. Li, J., Rozier, K.Y., Pu, G., Zhang, Y., Vardi, M.Y.: SAT-Based Explicit LTLf Satisfiability Checking. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019. pp. 2946–2953. AAAI Press (2019), <https://doi.org/10.1609/aaai.v33i01.33012946>
 30. Li, J., Vardi, M.Y., Rozier, K.Y.: Satisfiability checking for mission-time LTL. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15–18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 3–22. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_1
 31. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL satisfiability checking revisited. In: Proceedings of the 20th International Symposium on Temporal Representation and Reasoning, Pensacola, FL, USA, 2013. pp. 91–98. IEEE Computer Society (2013). <https://doi.org/10.1109/TIME.2013.19>
 32. Liaquat, D., de Lara, E.: The COVIDFree@Home website., <https://covidfreeathome.org/>
 33. Liu, M.: Overview of Datalog Extensions. In: Fraternali, P., Geske, U., Ruiz, C., Seipel, D. (eds.) Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming (DDL'98). In Conjunction with JICSLP'98. GMD Report, vol. 22, pp. 99–112 (1998)
 34. Mattarei, C., Cimatti, A., Gario, M., Tonetta, S., Rozier, K.Y.: Comparing different functional allocations in automated air traffic control design. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design (FMCAD'2015), Austin, Texas, USA. pp. 112–119. IEEE (2015)
 35. Mohapatra, A., Genesereth, M.: Aggregation in datalog under set semantics. Tech. rep., Tech. rep. 2012. url: <http://logic.stanford.edu/reports/LG-2012-01.pdf> (2012)
 36. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2008), Budapest, Hungary. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
 37. Narodytska, N., Bacchus, F.: Maximum satisfiability using core-guided maxsat resolution. In: Brodley, C.E., Stone, P. (eds.) Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27–31, 2014, Québec City, Québec, Canada. pp. 2717–2723. AAAI Press (2014), <http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8513>
 38. Pradella, M., Morzenti, A., Pietro, P.S.: Bounded satisfiability checking of metric temporal logic specifications. *ACM Trans. Softw. Eng. Methodol.* **22**(3), 20:1–20:54 (2013). <https://doi.org/10.1145/2491509.2491514>
 39. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Bosnacki, D., Edelkamp, S. (eds.) Proceedings of the 14th International Workshop on Model Checking Software (SPIN'07), Berlin, Germany. Lecture Notes in Computer Science, vol. 4595, pp. 149–167. Springer (2007). https://doi.org/10.1007/978-3-540-73370-6_11

40. Seo, J., Guo, S., Lam, M.S.: Socialite: Datalog extensions for efficient social network analysis. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). pp. 278–289. IEEE (2013)
41. Seo, J., Park, J., Shin, J., Lam, M.S.: Distributed socialite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment* **6**(14), 1906–1917 (2013)
42. Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., Zaniolo, C.: Big Data Analytics with Datalog Queries on Spark. In: *Proceedings of the 2016 International Conference on Management of Data*. pp. 1135–1149 (2016)
43. Yaman, S.G., Burholt, C., Jones, M., Calinescu, R., Cavalcanti, A.: Specification and validation of normative rules for autonomous agents. In: Lambers, L., Uchitel, S. (eds.) *Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings*. *Lecture Notes in Computer Science*, vol. 13991, pp. 241–248. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_13, https://doi.org/10.1007/978-3-031-30826-0_13

Appendix

Sec. A provides the correctness proof for the global lower bound (GLB), local upper bound (LUB) and G_A (Alg. 2). Sec. B provides the proof for the correctness of FOL⁺⁺ under-approximation encoding (Thm. 3). Sec. C illustrates the over- and under-approximation of the summation. Sec. D illustrates the algorithm TOOL. Sec. E shows how the monitorable fragment of MFOTL_ω can be captured by FOL⁺⁺. Sec. F describes the approach used to model aggregation constraints in FOL*. Finally, Sec. G provides details on how TOOL supports incremental solving for under-approximation.

A Correctness Proof for GLB and LUB

In this section, we provide the full proof of Lemma 1 and Thm. 1. We prove the lemmas together by induction (in the layer where the target sum is stratified). We prove the base case in Sec. A.1, the inductive step in Sec. A.2 and conclude the proof in Sec. A.3.

A.1 Base case proof

First, we consider the base case where the target summation is stratified at layer 0. The base cases are stated as Lemmas 2, 3 and 4.

Lemma 2 (Local correctness of GLB and LUB at layer 0). *Suppose $\text{Sum}(S, p, \text{val})$, denoted sum^0 , is a stratified summation at level 0 (see Def. 1) and D_\downarrow is a domain. Let $\text{LUB}(\text{sum}^0, D_\downarrow)$ and $\text{GLB}(\text{sum}^0, D_\downarrow)$ be the local upper-bound and global lower-bound of sum^0 , respectively. Let $\text{sum}_{D_\downarrow}^0$ be the under-approximation of sum^0 in D_\downarrow ($\text{sum}_{D_\downarrow}^0 = \sum_{s \in D_\downarrow} \text{ite}(s.\text{ext} \wedge p(s), \text{val}(s), 0)$). The following statement is true:*

$$\text{GLB}(\text{sum}^0, D_\downarrow) \leq \text{sum}_{D_\downarrow}^0 \leq \text{LUB}(\text{sum}^0, D_\downarrow)$$

Proof. First, we prove the inequality $\text{GLB}(\text{sum}^0, D_\downarrow) \leq \text{sum}_{D_\downarrow}^0$ by contradiction. Suppose $\text{GLB}(\text{sum}^0, D_\downarrow) > \text{sum}_{D_\downarrow}^0$, then it is the case that

$$\sum_{s \in S_{D_\downarrow}} \text{ite}(\neg s.\text{ext} \vee \text{G_A}(\neg p(s), D_\downarrow), 0, \text{GLB}(\text{val}(s), D_\downarrow)) > \sum_{s \in D_\downarrow} \text{ite}(s.\text{ext} \wedge p(s), \text{val}(s), 0)$$

Since sum^0 is stratified at level 0, $p(s)$ and $\text{val}(s)$ do not have summation in them (by Def. 1). Therefore, $\text{G_A}(\neg p(s), D_\downarrow) = \text{G}(\neg p(s), D_\downarrow)$. Feng et al. [22] proved (Lemma 3) that $\text{G}(\neg p(s), D_\downarrow)$ is an over-approximation of $\neg p(s)$ (i.e., $\neg p(s) \Rightarrow \text{G}(\neg p(s), D_\downarrow)$), hence $\neg \text{G}(\neg p(s), D_\downarrow) \Rightarrow p(s)$. Therefore, for every $s \in S$, if s contributes $\text{GLB}(\text{val}(s), D_\downarrow)$ to $\text{GLB}(\text{sum}^0, D_\downarrow)$, then it must contribute $\text{val}(s)$ to $\text{sum}_{D_\downarrow}^0$. Since $\text{val}(s)$ does not contain any summation (sum^0 is stratified at layer 0), it is easy to see that $\text{GLB}(\text{val}(s), D_\downarrow) = \text{val}(s)$. Therefore,

$$\sum_{s \in S_{D_{\downarrow}}} ite(\neg s.ext \vee G_A(\neg p(s), D_{\downarrow}), 0, GLB(val(s), D_{\downarrow})) \leq \sum_{S_{D_{\downarrow}}}^s ite(s.ext \wedge p(s), val(s), 0)$$

This reaches a contradiction.

Second, we prove the inequality $sum_{D_{\downarrow}}^0 \leq LUB(sum^0, D_{\downarrow})$ by contradiction: Suppose $sum_{D_{\downarrow}}^0 > LUB(sum^0, D_{\downarrow})$, then it is the case that

$$\sum_{s \in S_{D_{\downarrow}}} ite(s.ext \wedge G(p(s), D_{\downarrow}), LUB(val(s), D_{\downarrow}), 0) \leq \sum_{S_{D_{\downarrow}}}^s ite(s.ext \wedge p(s), val(s), 0)$$

Since sum^0 is stratified at level 0, the result of $p(s)$ and $val(s)$ will not have summation in them. Therefore, $G_A(\neg p(s), D_{\downarrow}) = G(\neg p(s), D_{\downarrow})$. Feng et al. proved (Lemma 3 in [22]) that $G(p(s), D_{\downarrow})$ is an over-approximation of $p(s)$ (i.e., $p(s) \Rightarrow G(p(s), D_{\downarrow})$). Therefore, for every $s \in S$, if s contributes $val(s)$ to $sum_{D_{\downarrow}}^0$, then it must contribute $LUB(val(s), D_{\downarrow})$ to $LUB(sum^0, D_{\downarrow})$. Since $val(s)$ does not contain any summation (sum^0 is stratified at layer 0), it is easy to see that $GLB(val(s), D_{\downarrow}) = val(s)$. Therefore,

$$\sum_{s \in S_{D_{\downarrow}}} ite(s.ext \wedge G(p(s), D_{\downarrow}), LUB(val(s), D_{\downarrow}), 0) > \sum_{S_{D_{\downarrow}}}^s ite(s.ext \wedge p(s), val(s), 0)$$

This is a contradiction. Therefore, both inequalities are proven.

Lemma 3 (Global correctness GLB at layer 0). *Suppose $Sum(S, p, val)$, denoted as sum^0 , is a stratified summation at level 0 (see Def. 1) and D_{\downarrow} is a domain. Let $GLB(sum^0, D_{\downarrow})$ be the global lower-bound of sum^0 . For every domain such that $D \supseteq D_{\downarrow}$, let $sum_D = \sum_{S_D}^s ite(s.ext \wedge p(s), val(s), 0)$ be the under-approximation of sum^0 in D . The following relation always holds: $GLB(sum^0, D_{\downarrow}) \leq sum_D^0$.*

Proof. Proof by contradiction: suppose that there exists a domain $D \supseteq D_{\downarrow}$ such that $GLB(sum^0, D_{\downarrow}) > sum_D^{i+1}$. Since $D \supseteq D_{\downarrow}$, $sum_D^0 \geq sum_{D_{\downarrow}}^0$. By Lemma 2, $GLB(sum^0, D_{\downarrow}) \leq sum_{D_{\downarrow}}^0$, hence $GLB(sum^0, D_{\downarrow}) \leq sum_D^0$. Contradiction.

Before moving on to the inductive step, we establish an important lemma for the function G_A since it is used in the definition of LUB and GLB, and behaves differently from G when the input formula contains summations.

Lemma 4. *Suppose ϕ^0 is a FOL^{*+} formula where every summation in ϕ^0 is stratified at layer 0 (see Def. 1), and D_{\downarrow} is a domain. The grounded formula $G_A(sum^0, D_{\downarrow})$ (Alg. 2) is an over-approximation of ϕ^0 (i.e., if ϕ^0 is satisfiable, then $G_A(\phi^0, D_{\downarrow})$ is satisfiable).*

Proof. If ϕ^0 does not contain any summation, then $G_A(\phi^0, D_{\downarrow}) = G(\phi^0, D_{\downarrow})$, and Feng et al. [22] proved that $G(\phi^0, D_{\downarrow})$ is an over-approximation of ϕ^0 . If ϕ^0 contains a summation sum^0 , then it is encoded as a fresh integer variable i

(L: 13 of Alg. 2) subject to the constraint req_{sum} (Def. 6). It suffices to show that the range of i , $[GLB(sum^0, D_\downarrow),)$ includes the possible value of sum_D^0 for all $D \supseteq D_\downarrow$. By Lemma 3, $sum_D^0 \geq GLB(sum^0, D)$, and thus sum_D^0 is in $[GLB(sum^0, D_\downarrow),)$. Therefore, $G_A(\phi^0, D_\downarrow)$ is an over-approximation of ϕ^0 .

A.2 Inductive step

Now we prove the inductive step. First, we establish the inductive hypothesis.

Hypothesis 1 (Inductive hypothesis of GLB) *Let a domain D_\downarrow and a summation sum^i stratified at layer i be given. Then for every domain $D \supseteq D_\downarrow$, $GLB(sum^i, D_\downarrow) \leq sum_D^i$, where sum_D^i is the under-approximation of sum^i in domain D .*

Hypothesis 2 (Inductive hypothesis of LUB) *Let a domain D_\downarrow , and a summation sum^i stratified at layer i be given. Then $LUB(sum^i, D_\downarrow) \geq sum_{D_\downarrow}^i$ where $sum_{D_\downarrow}^i$ is the under-approximation of sum^i in D_\downarrow .*

Hypothesis 3 (Inductive hypothesis of G_A) *Given a domain D_\downarrow , and an FOL^{*+} formula ϕ^i whose summations are stratified at layer i , $G_A(\phi^i, D_\downarrow)$ is an over-approximation of ϕ^i .*

We now prove the inductive lemmas by assuming the above inductive hypotheses.

Lemma 5 (Inductive local correctness of GLB and LUB). *Suppose $Sum(S, p, val)$, denoted as sum^{i+1} , is a stratified summation at level $i+1$ (see Def.1) and D_\downarrow is a domain. Let $sum_{D_\downarrow}^{i+1}$ be the under-approximation of sum^{i+1} in D_\downarrow . If Hypotheses 1, 2 and 3 holds, then*

$$GLB(sum^{i+1}, D_\downarrow) \leq sum_{D_\downarrow}^{i+1} \leq LUB(sum^{i+1}, D_\downarrow)$$

Proof. First, we prove the inequality $GLB(sum^{i+1}, D_\downarrow) \leq sum_{D_\downarrow}^{i+1}$. By Def. 4, $GLB(sum^{i+1}, D_\downarrow) = \sum_{s \in S_{D_\downarrow}} ite(\neg s.ext \vee G_A(\neg p(s), D_\downarrow), 0, GLB(val(s), D_\downarrow))$. Since sum^{i+1} is stratified at layer $i+1$, by Def. 1, $\neg p(s)$ only contains summations that are stratified at layer i or below. Therefore, by Hypothesis 3, $G_A(\neg p(s), D_\downarrow)$ is an over-approximation of $\neg p(a)$, and hence if s contributes $GLB(val(s), D_\downarrow)$ to $GLB(sum^{i+1}, D_\downarrow)$, then it must also contribute $val(s)$ to $sum_{D_\downarrow}^{i+1}$. Therefore, we can show $GLB(val(s), D_\downarrow) \leq val(s)$ to prove $GLB(sum^{i+1}, D_\downarrow) \leq sum_{D_\downarrow}^{i+1}$. Now consider $GLB(val(s), D_\downarrow)$, By Def. 4, there are five cases:

- (1) if $val(s)$ is a constant or a variable, then $GLB(val(s), D_\downarrow) = val(s)$. \square
- (2) if $val(s) = -t$ then we create an obligation showing $LUB(val(s), D_\downarrow) \geq val(s)$. Without loss of generality (WLOG), we can assume that t does not contain any other negation operator, ‘ $-$ ’, since otherwise the negation on t can be pushed in.

- (3) if $val(s) = t_1 + t_2$, then we create an obligation to show $GLB(val(t_1), D_\downarrow) \leq t_1 \wedge GLB(val(t_2), D_\downarrow) \leq t_2$.
- (4) if $val(s) = c \times t$, WLOG, we can assume $c > 0$ (if $c < 0$, can rewrite it as $-(-c \times t)$), then we create an obligation to show $GLB(t, D_\downarrow) \leq t$.
- (5) if $val(s)$ is a summation, then by Hypothesis 1, $GLB(val(s), D_\downarrow) \leq val(s)$.

Cases (1) and (5) are terminal and have already been proven. Case (2) is a special terminal case where we need to prove $LUB(t, D_\downarrow) \geq t$ for some negation-free term t . Cases (3) and (4) are non-terminal cases which generate a set of new proof obligations. Since $val(s)$ is a finite expression, Cases (3) and (4) will reach one of the terminal cases (1), (2) or (5).

To prove Case (2): $LUB(t, D_\downarrow) \geq t$ for some negation-free term t , we consider the definition of LUB (Def. 5) which consists of five cases.

- (i) t is a constant or a variable. Then $GLB(t, D_\downarrow) = t$. \square
 - (ii) $t = -t'$. However, since we assumed that t does not contain negation, this case is unreachable.
 - (iii) $val(s) = t_1 + t_2$. Then we create an obligation to show $LUB(val(t_1), D_\downarrow) \geq t_1 \wedge LUB(val(t_2), D_\downarrow) \geq t_2$.
 - (iv) if $val(s) = c \times t$, WLOG, we can assume $c > 0$ (if $c < 0$, can rewrite it as $-(-c \times t)$). Then we create an obligation to show $LUB(t, D_\downarrow) \geq t$.
 - (v) if $val(s)$ is a summation, then by Hypothesis 2, $LUB(val(s), D_\downarrow) \geq val(s)$.
- \square

Cases (i) and (v) are terminal and Case (ii) is unreachable. Cases (iii) and (iv) are non-terminal, which generate more proof obligations. Given that t is a finite expression, by recursively analyzing the proof obligations, these cases will eventually reach either Case (i) or Case (v). This proves Case (2). \square

Combining Case (2) with Cases (1) and (3)-(5), we now have proven $GLB(sum^{i+1}, D_\downarrow) \leq sum_{D_\downarrow}^{i+1}$. Combining this with the proven fact that $G_A(\neg p(s), D_\downarrow) \Rightarrow \neg p(s)$, we obtain the first inequality $GLB(sum^{i+1}, D_\downarrow) \leq sum_{D_\downarrow}^{i+1}$. \square

The proof for the second inequality, $sum_{D_\downarrow}^{i+1} \leq LUB(sum^{i+1}, D_\downarrow)$, is identical to the proof of the first inequality with a few exceptions: (1) we prove that $G_A(p(s), D_\downarrow) \Rightarrow p(s)$ given Hypothesis 3; (2) we prove $LUB(val(s), D_\downarrow) \geq val(s)$ by case analysis following the definition of LUB (Def. 5), and (3) we prove $GLB(t, D_\downarrow) \leq t$ for any negation-free term t . Due to the similarity, the detailed proof is omitted.

Lemma 6 (Inductive global correctness GLB). *Suppose $Sum(S, p, val)$, denoted as sum^{i+1} , is a stratified summation at level $i + 1$ (see Def. 1) and D_\downarrow is a domain. Let $GLB(sum^0, D_\downarrow)$ be the global lower-bound of sum^{i+1} . For every domain $D \supseteq D_\downarrow$, let $sum_D = \sum_{S_D}^s ite(s.ext \wedge p(s), val(s), 0)$ be the under-approximation of sum^{i+1} in D . If Hypotheses 1, 2 and 3 hold, then the following relation always holds: $GLB(sum^{i+1}, D_\downarrow) \leq sum_D^{i+1}$.*

Proof. Proof by contradiction: suppose there exists a domain $D \supseteq D_\downarrow$ such that $GLB(sum^{i+1}, D_\downarrow) > sum_D^{i+1}$. Since $D \supseteq D_\downarrow$, $sum_D^{i+1} \geq sum_{D_\downarrow}^{i+1}$. By Lemma 5, $GLB(sum^{i+1}, D_\downarrow) \leq sum_{D_\downarrow}^{i+1}$, hence $GLB(sum^0, D_\downarrow) \leq sum_D^{i+1}$. \square

The following lemma is an inductive generalization to Lemma 4, which is necessary for induction for the Hypothesis 3.

Lemma 7. *Suppose ϕ^{i+1} is a FOL*⁺ formula where every summation in ϕ^{i+1} is stratified at layer $i + 1$ (see Def. 1), and D_\downarrow is a domain. If Hypotheses 1, 2 and 3 hold, then the grounded formula $G_A(sum^{i+1}, D_\downarrow)$ (Alg. 2) is an over-approximation of ϕ^{i+1} (i.e., if ϕ^{i+1} is satisfiable, then $G_A(\phi^{i+1}, D_\downarrow)$ is also satisfiable).*

Proof. Every summation \sum^{i+1} in ϕ^{i+1} is encoded as a fresh integer variable i (L: 13 of Alg. 2) subject to the constraint req_{sum} (Def. 6). It is sufficient to show that the range of i , $[GLB(sum^{i+1}, D_\downarrow), \dots]$ includes the possible value of sum_D^{i+1} for all $D \supseteq D_\downarrow$. By Lemma 6, $sum_D^{i+1} \geq GLB(sum^{i+1}, D)$, and thus sum_D^{i+1} is in $[GLB(sum^{i+1}, D_\downarrow), \dots]$. Therefore, $G_A(\phi^{i+1}, D_\downarrow)$ is an over-approximation of ϕ^{i+1} .

A.3 Proving correctness of GLB and LUB

Given the base cases Lemmas 2, 3 and 4, and the inductive step, Lemmas 5, 6 and 7, the proofs of the correctness of GLB, LUB (Lemma 1) and G_A (Thm. 1) are the direct result of induction.

B Proving Correctness of FOL*⁺ Under-approximation

In this section, we prove Thm. 2. The proof follows an inductive argument with the help of an addition lemma.

Lemma 8. *Let an FOL*⁺ formula ϕ and a domain of relational objects D_\downarrow be given. Let ϕ_g be the grounded encoding $G_A(\phi, D_\downarrow)$, and let ϕ_g^\perp be the formula $\phi_g \wedge Inc(\phi_g, D_\downarrow)$. For every summation $sum = Sum(S, p, val)$ in ϕ , in every satisfying solution v to ϕ_g^\perp , $v(sum) = v(sum_{D_\downarrow}) = v(GLB(sum, D_\downarrow)) = v(LUB(sum, D_\downarrow))$, where sum_{D_\downarrow} is the under-approximated aggregation in domain D_\downarrow , and GLB and LUB are the global lower-bound (Def. 4) and local upper-bound (Def. 5).*

Lemma 8 states that the numerical over-approximations on aggregations in ϕ_g collapse to their LUB, GLB, and standard semantic definition in ϕ_g^\perp . Once we prove Lemma 8, then we can prove Thm. 2 by showing that the propositional over-approximations also collapse, as demonstrated in Feng et al. [22].

We now prove Thm. 2 and Lemma 8 together by induction on the layer where aggregations are stratified.

Base case: We prove that Thm. 2 and Lemma 8 hold for a FOL*⁺ formula where every summation sum is stratified at level 0. For every summation sum , its over-approximation $G_A(arg, D_\downarrow)$ is an integer i subject to $bound_{arg}(sum)$:

$$(a) \quad i \geq GLB_{D_\downarrow}^{sum}$$

(b) if $i > \text{LUB}_{D_\downarrow}^{sum}$ then $\exists s' \cdot p(s') \wedge \text{val}(s') + \text{Sum}(S, \lambda s : s \neq s' \wedge p(s), \text{val}) = i$,

where

$$\begin{aligned} - \text{LUB}_{D_\downarrow}^{sum} &= \sum_{s \in S_{D_\downarrow}} \text{ite}(s.\text{ext} \wedge \text{G_A}(p(s), D_\downarrow), \text{LUB}(\text{val}(s), D_\downarrow), 0) \text{ and} \\ - \text{GLB}_{D_\downarrow}^{sum} &= \sum_{s \in S_{D_\downarrow}} \text{ite}(\neg s.\text{ext} \vee \text{G_A}(\neg p(s), D_\downarrow), 0, \text{GLB}(\text{val}(s), D_\downarrow)). \end{aligned}$$

Since sum is stratified at level 0, for every $s \in S_{D_\downarrow}$, $p(s)$ and $\text{val}(s)$ do not contain summations. It has been established by Feng et al. [22] that $\text{G_A}(p(s), D_\downarrow) \wedge \text{Inc}(\phi_g, D_\downarrow)$ is eq-satisfiable as $p(s)$ in D_\downarrow , thus leading to $\text{G_A}(p(s), D_\downarrow) \iff \neg \text{G_A}(\neg p(s), D_\downarrow)$. As for $\text{val}(s)$, based on Def. 4 and 5, it can be shown that $\text{LUB}(\text{val}(s), D_\downarrow) = \text{GLB}(\text{val}(s), D_\downarrow) = \text{val}(s)$.

Therefore, we can conclude that $v(\text{LUB}_{D_\downarrow}^{sum}) = v(\text{GLB}_{D_\downarrow}^{sum})$ for every satisfying solution v to ϕ_g^\perp . Leveraging the correctness of GLB and LUB (as per Lemma 2 and Lemma 3), the value $v(sum_{D_\downarrow})$ must be bounded both above and below by $v(\text{LUB}_{D_\downarrow}^{sum})$ and $v(\text{GLB}_{D_\downarrow}^{sum})$, respectively. Consequently, we can assert that $v(sum_{D_\downarrow}) = v(\text{LUB}_{D_\downarrow}^{sum}) = v(\text{GLB}_{D_\downarrow}^{sum})$.

Finally, considering $\text{bound}_{arg}(sum)$, the value of i is also bounded below by $v(\text{GLB}_{D_\downarrow}^{sum})$. Furthermore, i is bounded above by $v(\text{LUB}_{D_\downarrow}^{sum})$ unless there exists s' such that $p(s') \wedge \text{Sum}(S, \lambda s : s \neq s' \wedge p(s), \text{val})$. However, due to $\text{Inc}(\phi_g, D_\downarrow)$, such s' does not exist since it must be semantically different from every $s \in S_{D_\downarrow}$.

Therefore, we can conclude that $v(i) = v(sum_{D_\downarrow}) = v(\text{LUB}_{D_\downarrow}^{sum}) = v(\text{GLB}_{D_\downarrow}^{sum})$. Hence, we have established the base case for Lemma 8. The proof for the base case of Thm. 2 follows in a similar way. \square

Inductive generalization step: We assume that Thm. 2 and Lemma 8 hold for any FOL^{*+} formula where every summation sum is stratified at level i or below (**inductive hypothesis**).

To prove Thm. 2 and Lemma 8 hold for any FOL^{*+} formula where every summation sum is stratified at level $i+1$, we follow the same proof structure as the proof for the base case. We first prove that

$$v(\text{LUB}_{D_\downarrow}^{sum}) = v(\text{GLB}_{D_\downarrow}^{sum}) \text{ for any solution } v \text{ to } \phi_g^\perp, \text{ where}$$

$$\begin{aligned} - \text{LUB}_{D_\downarrow}^{sum} &= \sum_{s \in S_{D_\downarrow}} \text{ite}(s.\text{ext} \wedge \text{G_A}(p(s), D_\downarrow), \text{LUB}(\text{val}(s), D_\downarrow), 0) \text{ and} \\ - \text{GLB}_{D_\downarrow}^{sum} &= \sum_{s \in S_{D_\downarrow}} \text{ite}(\neg s.\text{ext} \vee \text{G_A}(\neg p(s), D_\downarrow), 0, \text{GLB}(\text{val}(s), D_\downarrow)). \end{aligned}$$

It is sufficient to prove the following two conditions: (1) $\text{G_A}(p(s), D_\downarrow) \iff \neg \text{G_A}(\neg p(s), D_\downarrow)$ is a logical consequence of ϕ_g^\perp , and (2) $v(\text{GLB}(\text{val}(s), D_\downarrow)) = v(\text{LUB}(\text{val}(s), D_\downarrow))$ for any solution v to ϕ_g^\perp . Conditions (1) and (2) follow from the inductive hypothesis on Thm. 2 and Lemma 8, respectively.

From the proven fact $v(\text{LUB}_{D_\downarrow}^{sum}) = v(\text{GLB}_{D_\downarrow}^{sum})$, we can show that Thm. 2 and Lemma 8 hold for ϕ with aggregations stratified at layer $i+1$. \square

Conclusion: Thm. 2 holds for the base case as well as the inductive generalization step. Therefore, Thm. 1 holds for every FOL^{*+} formula ϕ . \square

C Example of Summation Over-and Under-approximation

Consider the property P_1 described in Fig. 1a. P_1 contains the condition $C1$ that compares the size of user transactions with their daily spending of the last 7 days. The daily transaction amount made by a user u at day t is expressed as a summation sum in the form:

$$Sum(Trans, \lambda tr' : tr'.u = u \wedge tr'.time = t, \lambda tr' : tr'.x).$$

Suppose D_\downarrow is domain consistent of three relational objects of the class $Trans$ $\{tr_1, tr_2, tr_3\}$.

- The under-approximation sum_D is
 $ite(tr_1.u = u \wedge tr_1.time = t, tr_1.x, 0) + ite(tr_2.u = u \wedge tr_2.time = t, tr_2.x, 0) + ite(tr_3.u = u \wedge tr_3.time = t, tr_3.x, 0)$.
- The over-approximation of sum is a fresh integer value i such that
 $i \geq GLB(sum, D_\downarrow)$ and $i \geq GLB(sum, D_\downarrow) \implies \exists tr : Trans \cdot i = tr.x + Sum(Trans, \lambda tr' : tr'.u = u \wedge tr'.time = t \wedge tr \neq tr', \lambda tr' : tr'.x)$.

Since sum is stratified at layer-0, $GLB(sum, D_\downarrow) = GLB(sum, D_\downarrow) = sum_{D_\downarrow}$ (Cor. 1). Now consider sum' in the form of

$$Sum(Trans, \lambda tr.x \geq sum, \lambda tr' : tr'.x).$$

- The under-approximation sum'_D is
 $ite(tr_1.x \geq sum_{D_\downarrow}, tr_1.x, 0) + ite(tr_2.x \geq sum_{D_\downarrow}, tr_2.x, 0) + ite(tr_3.x \geq sum_{D_\downarrow}, tr_3.x, 0)$.
- The over-approximation of sum' is a fresh integer value i' such that
 $i' \geq GLB(sum', D_\downarrow)$ and $i' \geq GLB(sum', D_\downarrow) \implies \exists tr : Trans \cdot i' = tr.x + Sum(Trans, \lambda tr' : tr'.x \geq sum \wedge tr \neq tr', \lambda tr' : tr'.x)$.

Let i be the over-approximation of sum defined above:

- The global lower-bound $GLB(sum', D_\downarrow)$ is
 $ite(tr_1.x < i, 0, tr_1.x) + ite(tr_2.x < i, 0, tr_2.x) + ite(tr_3.x < i, 0, tr_3.x)$
- The local upper-bound $LUB(sum', D_\downarrow)$ is
 $ite(tr_1.x \geq i, tr_1.x, 0) + ite(tr_2.x \geq i, tr_2.x, 0) + ite(tr_3.x \geq i, tr_3.x, 0)$.

D Illustration of Running TOOL

Consider a banking system that supports transfer between users. A user u can transfer x units of money to a different user v by transfer: $Trans(u, v, x)$.

The bank establishes the following requirements: (R1) A user can transfer at most 5000 units of money every day. (R2) If a single transfer is for an amount greater than 1000 units, then the user who initiated this transfer must have transferred out 3000 units on the previous day. (R3) The banking system only accepts refund requests for transfers under 1000 units of money. Inspired by the

legal property P_1 in Fig. 1, we introduce a new property P_2 adapted for the banking industry: For any transfer with amount more than 1000 units, the transfer amount should not be higher than the usual transferer's total daily spending over the last 7 days. We formalize P_2 's negation as: $\neg P_2 = \exists tr : Trans \cdot (tr.x > 3000 \wedge (\forall t : Time \cdot trans.time - 7 \leq t < trans.time \Rightarrow Sum(Trans, \lambda tr' : tr'.u = tr.u \wedge tr.time = t, \lambda tr' : tr'.x < tr.x)))$. We now illustrate SEARCH-A. For each iteration, we denote ϕ_g and ϕ_g^\perp as the over- and under-approximation queries computed on L: 4 and L: 5 of Alg. 1, respectively. We write sum^\uparrow and sum^\downarrow as the over- and under-approximation of sum , respectively. Note that for the sum $Sum(Trans, \lambda tr' : tr'.u = tr.u \wedge tr.time = t, \lambda tr' : tr'.x < tr.x))$, its GLB (Def. 4), LUB (Def. 5) and sum^\downarrow always have the same value in any domain because the filtering function and value function do not return expressions with sums (Cor. 1). Therefore, we use sum^\downarrow to represent all of them.

Iteration 1. $D_\downarrow = \emptyset$. The over-approximation ϕ_g introduces a relational object tr^1 due to $\neg p$ where $tr^1.x > 1000$. ϕ_g is satisfiable, but ϕ_g^\perp is not (solved by the SMT solver Z3 [15] using techniques from [?]) since $tr_1 \notin D_\downarrow$. Therefore, D_\downarrow is expanded by adding tr^1 .

Iteration 2. $D_\downarrow = \{tr^1\}$. ϕ_g introduces a new summation $sum_1^\uparrow = 3000$ due to (R2) because the sum of transfer by $tr^1.u$ at $tr^1.time - 1$ is 3000. ϕ_g is satisfiable, but ϕ_g^\perp is UNSAT since $0 = sum_1^\downarrow \neq sum_1^\uparrow$. Therefore, D_\downarrow is expanded by adding sum_1 as a summation object.

Iteration 3. $D_\downarrow = \{tr^1, sum_1\}$. ϕ_g introduces a relational object tr^2 and a new sum sum_2 due to req_{sum} where $tr^2.u = tr^1.u$, $tr^2.time = tr^1.time - 1$ and $sum_2^\uparrow + tr^2.x = sum_1^\uparrow$. ϕ_g is satisfiable, but ϕ_g^\perp is not since $tr_2 \notin D_\downarrow$. We assume that D_\downarrow is expanded by tr^2 .

Iteration 4. $D_\downarrow = \{tr^1, sum_1, tr^2\}$. ϕ_g introduces a new summation $sum_3^\uparrow = 3000$ because R2 describes the sum of transfer amounts by $tr^2.u$ at $tr^2.time - 1$ when $tr^2.x > 1000$. ϕ_g is satisfiable, but ϕ_g^\perp is not since $sum_3 \downarrow = 0$ or $sum_2^\downarrow + tr^2.x < 3000$. Suppose that D_\downarrow is expanded by adding sum_2 to D_\downarrow .

Iterations 5 - 10. $D_\downarrow = \{tr^1, sum_1, tr^2, sum_2\}$. Suppose that the domain expansion follows a process similar to that of iterations 3-4. At the end of iteration 10, there are 4 relational objects, tr^2, tr^3, tr^4, tr^5 , all of which occurred one day before $tr^1.time$ and are initiated by $tr^1.u$.

The final iteration. $D_\downarrow = \{tr^1, sum_1, tr^2, sum_2, tr^3, sum_4, \dots, tr^5\}$. ϕ_g^\perp becomes satisfiable with the solution $tr^1.x = 5000$, $tr^1.u = 0$, $tr^1.time = 1$, $tr^2.x = tr^3.x = tr^4.x = tr^5.x = 800$, $tr^2.u = tr^3.u = tr^4.u = tr^5.u = 0$ and $tr^2.time = tr^3.time = tr^4.time = tr^5.time = 0$. Therefore, $\neg P_2$ is satisfied implying that property P_2 might be violated.

On the other hand, if R1 is tightened to restrict a user from transferring more than 3,000 units of money per day, then $\neg P_2 \wedge R1 \wedge R2$ is UNSAT.

E Capturing Monitorable MFOTL $_{\omega}$ with FOL $^{*+}$

In this section, we show that the monitorable fragment of MFOTL $_{\omega}$ can be captured by FOL $^{*+}$. First, we briefly describe MFOTL $_{\omega}$, and then discuss the relationship between FOL $^{*+}$ and MFOTL $_{\omega}$.

Metric first order temporal logic with aggregation (MFOTL $_{\omega}$) [9].

Syntax. A *signature* S is a tuple (F, R, ι) , where F is a set of function symbols and R is a finite set of predicate symbols (for relation) disjoint from F , respectively. The function $\iota : F \cup R \rightarrow \mathbb{N}$ associates each predicate symbol $r \in F \cup R$ with an arity $\iota(r) \in \mathbb{N}$. Let Var be a countable infinite set of variables from domain \mathbb{Z} , where $Var \cap (F \cup R) = \emptyset$. *Constants* are function symbols of arity 0, and $C \subset F$ is the set of constants in S . A *term* t is defined inductively as $t : c \mid v \mid t + t' \mid c \times t \mid f(t_1, \dots, t_n)$. Let $fv(t)$ be the set of the variables that occur in the term t . Let \bar{t} be a vector of terms, \bar{t}_x^k be a vector that contains x at index k , and \bar{t}_g be the set of ground terms. A *substitution* sub is a function from variables to terms.

The syntax of MFOTL $_{\omega}$ formulas is defined as follows: (1) \top and \perp , representing values “true” and “false”; (2) $t = t'$ and $t > t'$, for terms t and t' ; (3) $r(t_1 \dots t_{\iota(r)})$ for $r \in R$ and terms $t_1 \dots t_{\iota(r)}$; (4) $\phi \wedge \psi$, $\neg \phi$ for MFOTL $_{\omega}$ formulas ϕ and ψ ; (5) $(\exists x. \phi)$; (6) $\phi \mathcal{U}_I \psi$ (until), $\phi \mathcal{S}_I \psi$ (since), $\bullet_I \phi$ (next), $\circ_I \phi$ (previous) for MFOTL $_{\omega}$ formulas ϕ and ψ , and an interval $I \in \mathbb{I}$; and (7) $[\omega_t \bar{z}. \phi](y; \bar{g})$, an *aggregation formula*, where ω range over the elements in R, \bar{t}, \mathbb{I} , and a finite set of aggregation operators Ω , \bar{g} are the attributes on which grouping is performed and as \bar{z} , ranges over sequences of elements in Var , t is the term in which the aggregation operator ω is applied, and y is the attribute that stores the result and ranges over elements in Var . The variables in \bar{z} are ψ 's attributes that do not appear in the described relation.

The set of free variables of an aggregation formula, $[\omega_t \bar{z}. \omega]$, is defined as $fv([\omega_t \bar{z}. \omega]) := \{y\} \cup \bar{g}$. A variable is *bound* if it is not free. We denote by $fv(\phi)$ the sequence of free variables of a formula ϕ that is obtained by ordering the free variables of ϕ by their occurrence when reading the formula from left to right. A formula is *well-formed* if, for each of its aggregation subformulas, it holds that (i) $y \notin \bar{g}$, (ii) $fv(t) \subseteq fv(\psi)$, (iii) the elements of \bar{z} and \bar{g} are pairwise distinct, and (iv) $\bar{z} = fv(\psi) \bar{g}$.

Capturing MFOTL $_{\omega}$. The monitorable fragment of MFOTL $_{\omega}$ guarantees finiteness: for every solution, the number of tuples that hold for every relation at every time point is finite. To ensure finiteness, Basin et al. [9] uses a set of deviation rules to define the monitorable fragment. At a high level, the derivation rules ensure that every universally quantified variable that appears in a formula is always grounded by some finite relation (i.e., relations that hold for a finite number of tuples). For example, the formula $\forall xy R_1(x, y) \Rightarrow R_2(x, y)$ is monitorable if R_1 is a finite relation. On the other hand, the formula $\forall xy R_3(x) \Rightarrow R_2(x, y)$ is not monitorable since y is not guarded.

case1 :	$T(t = t', \tau_i)$	\rightarrow	$t = t'$
case2 :	$T(t > t', \tau_i)$	\rightarrow	$t > t'$
case3 :	$T(r(t_1, \dots, t_{\ell(r)}), \tau_i)$	\rightarrow	$\exists o : r \cdot \bigwedge_{j=1}^{\ell(r)} (o.j = t_j) \wedge (\tau_i = o.time)$
case4 :	$T(\neg \phi, \tau_i)$	\rightarrow	$\neg T(\phi, \tau_i)$
case5 :	$T(\phi \wedge \psi, \tau_i)$	\rightarrow	$T(\phi, \tau_i) \wedge T(\psi, \tau_i)$
case6 :	$T(\forall x \cdot r(\vec{t}_x^k) \Rightarrow \phi, \tau_i)$	\rightarrow	$\forall o : r \cdot T((r(\vec{t}_x^k) \Rightarrow \phi)[x \rightarrow o[k]], \tau_i)$
case7 :	$T(\bullet_I \phi, \tau_i)$	\rightarrow	$\exists o : TP \cdot \text{NEXT}(o.time, \tau_i) \wedge T(\phi, o.time) \wedge (o.time - \tau_i) \in I$
case8 :	$T(\circ_I \phi, \tau_i)$	\rightarrow	$\exists o : TP \cdot \text{PREV}(o.time, \tau_i) \wedge T(\phi, o.time) \wedge (\tau_i - o.time) \in I$
case9 :	$T(\phi \mathcal{U}_I \psi, \tau_i)$	\rightarrow	$\exists o : TP \cdot (o.time \geq \tau_i \wedge (o.time - \tau_i) \in I \wedge T(\psi, o.time))$ and $\forall o' : TP \cdot o'.time \cdot (\tau_i \leq o'.time < o.time \Rightarrow T(\phi, o'.time))$
case10 :	$T(\phi \mathcal{S}_I \psi, \tau_i)$	\rightarrow	$\exists o : TP \cdot (o.time \leq \tau_i \wedge (\tau_i - o.time) \in I \wedge T(\psi, o.time))$ and $\forall o' : TP \cdot (\tau_i \geq o'.time > o.time \Rightarrow T(\phi, o'.time))$
case11 :	$T([Sum_t \vec{z}. \psi](y : \vec{g}), \tau_i)$	\rightarrow	$y = \text{Sum}(s : Sol_\psi, t[\vec{z} \leftarrow s.\vec{z}; \vec{g} \leftarrow s.\vec{g}], T(\psi[\vec{z} \leftarrow s.\vec{z} \wedge s.\vec{g} = \vec{g}], \tau_i))$
case12 :	$T([Count_t \vec{z}. \psi](y : \vec{g}), \tau_i)$	\rightarrow	$y = \text{Count}(s : Sol_\psi, T(\psi[\vec{z} \leftarrow s.\vec{z} \wedge s.\vec{g} = \vec{g}], \tau_i))$
top :	$T(\phi)$	\rightarrow	$T(\phi, \tau_1)$

Fig. 3. Translation rules from MFOTL_ω to FOL^{*+}. TP is an internal class of relational objects used to represent time values at different time points. The predicate NEXT(t_1, t_2) (PREV(t_1, t_2)) asserts that t_1 is the next (previous) time value of t_2 . τ_1 is the value of the initial time point.

Lemma 9. *Let a MFOTL_ω formula ϕ be given. ϕ is monitorable only if we can find a logically equivalent guarded MFOTL_ω formula ϕ' such that every universally quantified variable x is grounded by some finite relations: $\forall x F(\vec{t}_x^k) \rightarrow (\psi(x))$ where F is a finite relation, \vec{t}_x^k is a vector of terms that contains a variable x at index k (i.e., x is a free variable in $\vec{t}_x^k[k]$), and ψ is a guarded MFOTL_ω formula.*

Lemma 9 can be proved by induction following the structure of the derivation rules in [9]. Note that the condition in Lemma 9 is a necessary condition for the monitorable fragment, and hence defines a superset of the fragment.

Now, we present the translation rules T for guarded MFOTL_ω (i.e., formula satisfies the guarded condition in Lemma 9) to FOL^{*+} in Tab. 3. The rules extend from the translation in [22] by defining the translation for aggregations (i.e., case11 and case12). Given an MFOTL_ω formula, the function T returns an FOL^{*+} formula $T(\phi)$ by following the structure of ϕ and applying T recursively to its components. Note that T assumes the universal quantifier in MFOTL_ω satisfies the guarded condition (i.e., $T(\forall x \cdot r(\vec{t}_x^k) \Rightarrow \phi, \tau_i)$ in case6).

Given an MFOTL formula $[Sum_t \vec{z}. \psi](y : \vec{g})$ and a time point τ_i , the translation function T (case11) creates an auxiliary relational class Sol_ψ where for every $s \in Sol_\psi$, s has attributes \vec{z} and \vec{g} . The translated FOL^{*+} formula defines the bag of relational objects $s : Sol_\psi$ that satisfies the formula $T(\psi[\vec{z} \leftarrow s.\vec{z}], \tau_i) \wedge s.\vec{g} = \vec{g}$ and then sums up the value of $t[\vec{z} \leftarrow s.\vec{z}; \vec{g} \leftarrow s.\vec{g}]$ for every relational object s in the bag. Finally, the sum is asserted to be equal to the value of y . The translation for count (case12) is defined analogously. Since ψ is a guarded MFOTL_ω formula, the number of solutions for ψ at time τ_i is finite. Therefore, Sol_ψ is a finite relation and can be represented with a finite number of relational objects.

Theorem 3 (Translation Correctness). *Let an MFOTL_ω formula ϕ that satisfies the guarded condition (Lemma 9) be given. For every FOL^{*+} formula*

$T(\phi)$, it has a finite solution (D, v) if and only if there exists a finite FO-temporal structure $(\vec{D}, \vec{\tau})$ and a valuation function v' such that $(\vec{D}, \vec{\tau}, v', 0) \models \phi$. Moreover, there exists a bijective mapping between the solutions of $T(\phi)$ and ϕ (solution-preserving).

Proof Sketch. The proof of Thm. 3 can be extended from the proof of Thm. 1 in [22], where a mapping function M (and its inverse M^{-1}) is defined to map every FOL⁺⁺ solution (D, v) to a finite FO-temporal structure $(\vec{D}, \vec{\tau})$ and a valuation function v' . We then demonstrate the correctness of this mapping in both directions: (1) $(D, v) \models T(\phi) \Rightarrow M(D, v) \models \phi$ and (2) $(\vec{D}, \vec{\tau}, v', 0) \models \phi \Rightarrow M^{-1}(\vec{D}, \vec{\tau}, v') \models T(\phi)$. [22] provides the definitions of M (and M^{-1}) and proof for *case1* to *case10*.

For the case of *Sum* (*case11*), we can prove it by induction with the inductive hypothesis: $T(\psi, \tau_i)$ is a solution-preserving translation of ψ at time point τ_i . We can then show that the set of relational objects $SOL := \{s : Sol_\psi \mid v(T(\psi[\vec{z} \leftarrow s.\vec{z}] \wedge s.\vec{g} = \vec{g}, \tau_i))\}$ represents exactly the set of tuples \vec{t} in $\llbracket \psi \rrbracket_{\vec{z}, v'}^{(\vec{D}, \vec{\tau}, i)}$: the set $\{v(s.\vec{z}) \mid s \in SOL\}$ is equivalent to $\llbracket \psi \rrbracket_{\vec{z}, v'}^{(\vec{D}, \vec{\tau}, i)}$. The equivalence follows from the inductive hypothesis: the set of values for \vec{z} that satisfies $T(\psi, \tau_i)$ is $\llbracket \psi \rrbracket_{\vec{z}, v'}^{(\vec{D}, \vec{\tau}, i)}$. Therefore, by substituting \vec{z} with $s.\vec{z}$ in ψ ($\psi[\vec{z} \leftarrow s.\vec{z}]$) and fixing the value of $s.\vec{g}$ to \vec{g} ($s.\vec{g} = \vec{g}$), we can prove the equivalence between $\{v(s.\vec{z}) \mid s \in SOL\}$ and $\llbracket \psi \rrbracket_{\vec{z}, v'}^{(\vec{D}, \vec{\tau}, i)}$. Then we can show that the aggregated sums have the same value since they follow the same aggregation semantics over finite sets, where the finiteness of the sets is guaranteed because ϕ is guarded. The case of *Count* ("case12") can be proved analogously. \square

Since FOL⁺⁺ can capture guarded MFOTL _{ω} formulas (according to Thm. 3), which is a superset of the monitorable fragment MFOTL _{ω} (by Lemma 9), we can conclude that FOL⁺⁺ can also capture the monitorable fragment MFOTL _{ω} .

F Modeling Aggregation in FOL*

Below we describe the approach used to model FOL⁺⁺ aggregation in FOL* following the recursive definition of aggregation [35].

Let S be a class of relational objects, $A \in \text{Sum}, \text{Count}, \text{Max}, \text{Min}$, p be a FOL⁺⁺ predicate, and val be an FOL⁺⁺ function. In any given domain D , the FOL⁺⁺ aggregation $A(S, p, val)$ captures a value by applying the aggregation function A over the bag (multi-set) of values defined by $val(o)$ for every relational object o in class S within the domain D that satisfies the predicate p . As the domain D is finite but not fixed, we model the aggregation in FOL* using the recursive definition of aggregation over a symbolic, finite, and ordered list.

To begin, we establish an ordering for relational objects of class S . We define a function, denoted by ord_S , which maps each relational object o of class S to a natural number with the following FOL* rules:

$$- \forall o, o' : S \cdot o \equiv o' \Rightarrow ord(o) = ord(o') \text{ (deterministic)}$$

- $\forall o : S \cdot \text{ord}(o) \geq 0$ (range)

Now that we have established an ordering for relational objects, we can proceed to define the aggregation rules:

- (1) $\text{Sum}(\emptyset) = 0$, $\text{Sum}(N + [x]) = \text{Sum}(N) + \mathbf{ite}(p(x), \text{val}(x), 0)$;
- (2) $\text{Count}(\emptyset) = 0$, $\text{Count}(N + [x]) = \text{Count}(N) + \mathbf{ite}(p(x), 1, 0)$;
- (3) $\text{Max}(\emptyset) = -\infty$, $\text{Max}(N + [x]) = \max(\text{Max}(N), \mathbf{ite}(p(x), \text{val}(x), -\infty))$; and
- (4) $\text{Min}(\emptyset) = \infty$, $\text{Min}(N + [x]) = \min(\text{Min}(N), \mathbf{ite}(p(x), \text{val}(x), \infty))$.

Since the list is ordered, the last relational object in the list (e.g., x) has the greatest order. Moreover, since the list captures all relational objects of class S in a domain where duplicates are collapsed, we can use the relational object with the greatest ordering to uniquely define a list whose last element is the object. Therefore, we can model the intermediate result of aggregation over any list with a new class of relational objects A_S (where A is the aggregation function) with two attributes: lst and value . The attribute lst is the order of the last relational object in the list which has been aggregated over, and value is the intermediate aggregation result. We can then define the aggregation rules in FOL*. We show the rule for Sum and the others can be defined analogously.

- $\forall s : \text{Sum}_S \cdot s.\text{lst} < 0 \Rightarrow s.\text{value} = 0$ (over an empty list)
- $\forall s : \text{Sum}_S \cdot s.\text{lst} \geq 0 \Rightarrow \exists s' : \text{Sum}_S \exists o : S(\text{ord}(o) = s.\text{lst} \wedge \mathbf{Pred}(s', s) \wedge s.\text{value} = s'.\text{value} + \mathbf{ite}(p(o), \text{val}(o), 0))$ (step backward consistency)
- $\mathbf{Pred}(s, s') = \neg(\exists s* : \text{Sum}_S \cdot s'.\text{lst} < s*.\text{lst} < s.\text{lst})$
- $\forall o : S \exists s : \text{Sum}_S \cdot s.\text{lst} = \text{ord}(o)$ (step forward consistency)
- $\forall s, s' : \text{Sum}_S s.\text{lst} = s'.\text{lst} \Rightarrow s.\text{value} = s'.\text{value}$ (aggregation determinism)

We can then capture the final aggregation result with the relational object s in A_S that contains the element with the greatest order:

$$\exists s : A_S \cdot \forall s' : A_S(s'.\text{lst} \leq s.\text{lst}) \wedge s.\text{value} = A(S, p, \text{val})$$

G TOOL Optimization: Incremental Solving

TOOL generates SMT queries for over-approximating and under-approximating satisfiability. Queries are generated with an incrementally increasing context which enables incremental solving to reuse learned clauses across different queries. We explain how to support incremental solving for over- and under-approximation below. While both TOOL and LEGOS support incremental solving for over-approximation, only TOOL supports incremental solving for under-approximation.

Incremental over-approximation. The grounded over-approximation queries (ϕ_g computed with the function G_A at L: 4 of Alg. 1) are monotonically strengthened with conjunctions during each iteration of the algorithm, allowing the underlying SMT solver to perform incremental solving. The monotonicity of over-approximation queries can be seen in two ways: (1) if a new requirement ψ is added to the set of requirements Ψ , then the grounded query is strengthened as $\text{G_A}(\Psi, D_\downarrow) \wedge \text{G_A}(\psi, D_\downarrow)$; (2) if the domain D_\downarrow is expanded to include a

new relational object o , then the ground query is strengthened as $G_A(\Psi, D_\downarrow) \wedge G_A(\Psi, D_\downarrow \cup o)$. Since ϕ_g is always strengthened with conjunctions, previously learned clauses are always valid, and supporting incremental solving is trivial.

Incremental under-approximation. The grounded under-approximation query ϕ_g^\perp (computed on L: 5 in Alg. 1) is the conjunction of two parts: (1) the over-approximation ϕ_g and (2) a domain inclusion constraint $Inc(D_\downarrow, \phi_g)$ that forces every relational object o' in ϕ_g to be semantically equivalent to some relational object o in D_\downarrow , i.e., $\bigvee_{o \in D_\downarrow} o' = r$. The domain inclusion constraint uses disjunction, and the number of disjunctive terms increases as D_\downarrow expands which makes incremental solving of the under-approximation non-trivial. This is because previous learned clauses might be invalidated by new disjunctive terms. To support incremental solving for this disjunction, we use an assumption literal l_{D_\downarrow} for each snapshot of D_\downarrow . When a domain D_\downarrow expands to D'_\downarrow , we formulate the domain inclusion constraint as a new rule: $l_{D'_\downarrow} \Rightarrow (l_{D_\downarrow} \vee Inc(D'_\downarrow \setminus D_\downarrow, \phi_g))$. The rule is added to the solver, and $l_{D'_\downarrow}$ is added as an assumption literal to the solver for solving the under-approximation in D'_\downarrow . To satisfy $l_{D'_\downarrow}$, the solver either asserts l_{D_\downarrow} or tries to satisfy $Inc(D'_\downarrow \setminus D_\downarrow, \phi_g)$. In the former case, the learned lesson from asserting l_{D_\downarrow} can be reused. To further improve the efficiency of incremental solving, we create an assumption literal for every class in the domain (instead of for the entire domain) and update it when a new relational object of the class is included in the domain. TOOL supports incremental under-approximation.

H TOOL Optimization: Relaxed Domain Expansion

One of the most expensive operations in the TOOL (Alg. 1) is the domain expansion. More specifically, if a FOL*+ formula's over-approximations (ϕ_g) is satisfiable while its under-approximation (ϕ_g^\perp) is unsatisfiable, then TOOL attempts to find the minimum solution (σ_m) to ϕ_g and add the relational objects from σ_m to the current domain D_\downarrow (L: 12 of Alg. 1). SEARCH-A computes the minimum solution using the UNSAT core-guided MaxSAT approach proposed in MaxRes [37] by minimizing (1) new instantiated relational objects in the over-approximation query ϕ_g and (2) existing relational objects in the under-approximated domain D_\downarrow . The minimality of σ_m ensures that the solution returned by the algorithm always has the smallest volume. Minimality also ensures the algorithm's correctness when B-UNSAT_{vb} is returned. However, we observed that the optimal domain-expansion strategy guided by σ_m is often costly and overly permissive. The cost of the σ_m computation correlates positively with the number of relational objects to be minimized, which *Increases* with the size of D_\downarrow due to (2) and significantly affects performance.

The domain expansion strategy can be relaxed to boost the algorithm's efficiency until a solution is found or the relaxed domain's size exceeds the given volume bound – a process called *boosting*. While boosting is not guaranteed to find the optimal solution, it is still useful when optimality is required, because optimal and non-optimal solutions might share relational objects, which could

be included in the domain during boosting. A relaxed domain expansion mode includes two new expansion strategies:

Greedy Best First Expansion (GBFE). Recall that the optimal expansion strategy used in SEARCH-A computes the optimal solution σ_m with the objective to minimize both (1) new instantiated relational objects in the over-approximation query ϕ_g and (2) existing relational objects in the under-approximated domain D_\downarrow . The cost of computing σ_m increases with the size of D_\downarrow due to (2). GBFE computes a relaxed minimum solution σ_m^* with the objective to only minimize new instantiated relational objects in ϕ_g (1). The solution σ_m^* corresponds to a minimum extension of D_\downarrow that satisfies ϕ_g . With GBFE, the number of relational objects to minimize does not always increase with the size of D_\downarrow .

Filtering by Object’s Activity Score. During domain expansion, newly instantiated relational objects in the over-approximation query ϕ_g are added to the domain D_\downarrow if they are in the relaxed minimum solution (σ_m^*) to the over-approximation ϕ_g . Since σ_{min}^* is computed using an UNSAT core-guided approach [37], if a relational object is not actively involved in the UNSAT cores for computing σ_m^* then it is unlikely to be included in σ_{min}^* . Based on this observation, we divide relational objects into “active” and “inactive” groups. A relational object is “active” when it is freshly instantiated with an activity score of 5. Each round of domain expansion decrements its activity score. If the activity score of a relational object is 0, it becomes “inactive”. On the other hand, if a relational object appears in an UNSAT core when computing σ_m^* , its activity score is reset to 5. The domain expansion first considers only the “active” group by disabling relational objects from the “inactive” group in σ_m^* . If the expansion fails i.e., σ_m^* does not exist), both groups are considered.