A Sorted Datalog Hammer for Supervisor Verification Conditions Modulo Simple Linear Arithmetic

Martin Bromberger[1](✉), Irina Dragoste[2], Rasha Faqeh[2], Christof Fetzer[2], Larry González[2], Markus Krötzsch[2], Maximilian Marx[2], Harish K Murali[1,3], and Christoph Weidenbach[1]

[1] Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
`{mbromber, weidenb}@mpi-inf.mpg.de`
[2] TU Dresden, Dresden, Germany
[3] IIITDM Kancheepuram, Chennai, India

Abstract. In a previous paper, we have shown that clause sets belonging to the Horn Bernays-Schönfinkel fragment over simple linear real arithmetic (HBS(SLR)) can be translated into HBS clause sets over a finite set of first-order constants. The translation preserves validity and satisfiability and it is still applicable if we extend our input with positive universally or existentially quantified verification conditions (conjectures). We call this translation a Datalog hammer. The combination of its implementation in SPASS-SPL with the Datalog reasoner VLog establishes an effective way of deciding verification conditions in the Horn fragment. We verify supervisor code for two examples: a lane change assistant in a car and an electronic control unit of a supercharged combustion engine. In this paper, we improve our Datalog hammer in several ways: we generalize it to mixed real-integer arithmetic and finite first-order sorts; we extend the class of acceptable inequalities beyond variable bounds and positively grounded inequalities; and we significantly reduce the size of the hammer output by a soft typing discipline. We call the result the sorted Datalog hammer. It not only allows us to handle more complex supervisor code and to model already considered supervisor code more concisely, but it also improves our performance on real world benchmark examples. Finally, we replace the before file-based interface between SPASS-SPL and VLog by a close coupling resulting in a single executable binary.

1 Introduction

Modern dynamic dependable systems (e.g., autonomous driving) continuously update software components to fix bugs and to introduce new features. However, the safety requirement of such systems demands software to be safety certified before it can be used, which is typically a lengthy process that hinders the dynamic update of software. We adapt the *continuous certification* approach [17] for variants of safety critical software components using a *supervisor* that guarantees important aspects through *challenging*, see Fig. 1. Specifically, multiple processing units run in parallel – *certified* and *updated not-certified* variants that produce output as *suggestions* and *explications*. The supervisor compares the behavior of variants and analyses their explications. The supervisor itself consists of a rather small set of rules that can be automatically verified and run by a reasoner such as SPASS-SPL. In this paper we concentrate on the further development of our verification approach through the sorted Datalog hammer.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 480–501, 2022.
https://doi.org/10.1007/978-3-030-99524-9_27

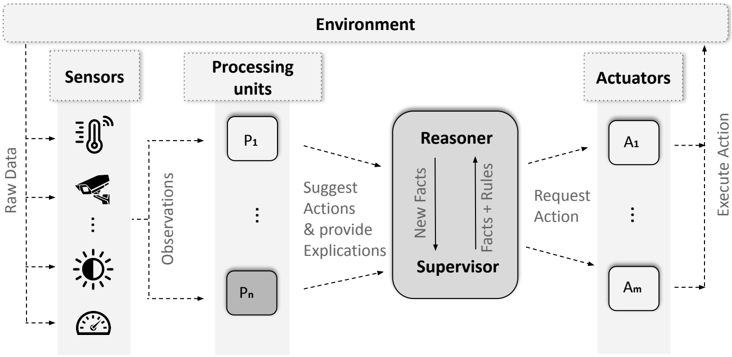

Fig. 1. The supervisor architecture.

While supervisor safety conditions formalized as existentially quantified properties can often already be automatically verified, conjectures about invariants requiring universally quantified properties are a further challenge. Analogous to the Sledgehammer project [8] of Isabelle [31] that translates higher-order logic conjectures to first-order logic (modulo theories) conjectures, our sorted Datalog hammer translates first-order Horn logic modulo arithmetic conjectures into pure Datalog programs, which is equivalent to the Horn Bernays-Schönfinkel clause fragment, called HBS.

More concretely, the underlying logic for both formalizing supervisor behavior and formulating conjectures is the hierarchic combination of the Horn Bernays-Schönfinkel fragment with linear arithmetic, HBS(LA), also called *Superlog* for Supervisor Effective Reasoning Logics [17]. Satisfiability of BS(LA) clause sets is undecidable [15,23], in general, however, the restriction to simple linear arithmetic BS(SLA) yields a decidable fragment [19,22].

Inspired by the test point method for quantifier elimination in arithmetic [27] we show that instantiation with a finite number of values is sufficient to decide whether a universal or existential conjecture is a consequence of a BS(SLA) clause set.

In this paper, we improve our Datalog hammer [11] for HBS(SLA) in three directions. First, we modify our Datalog hammer so it also accepts other sorts for variables besides reals: the integers and arbitrarily many finite first-order sorts $\mathcal{F}_i$. Each non-arithmetic sort has a predefined finite domain corresponding to a set of constants $\mathbb{F}_i$ for $\mathcal{F}_i$ in our signature. Second, we modify our Datalog hammer so it also accepts more general inequalities than simple linear arithmetic allows (but only under certain conditions). In [11], we have already started in this direction by extending the input logic from pure HBS(SLA) to pure positively grounded HBS(SLA). Here we establish a soft typing discipline by efficiently approximating potential values occurring at predicate argument positions of all derivable facts. Third, we modify the test-point scheme that is the basis of our Datalog hammer so it can exploit the fact that not all all inequalities are connected to all predicate argument positions.

Our modifications have three major advantages: first of all, they allow us to express supervisor code for our previous use cases more elegantly and without any additional preprocessing. Second of all, they allow us to formalize supervisor code that would have been out of scope of the logic before. Finally, they reduce the number of required test points, which leads to smaller transformed formulas that can be solved in much less time.

For our experiments of the test point approach we consider again two case studies. First, verification conditions for a supervisor taking care of multiple software variants of a lane change assistant. Second, verification conditions for a supervisor of a supercharged combustion engine, also called an ECU for Electronical Control Unit. The supervisors in both cases are formulated by $\mathrm{BS(SLA)}$ Horn clauses. Via our test point technique they are translated together with the verification conditions to Datalog [1] (HBS). The translation is implemented in our Superlog reasoner SPASS-SPL. The resulting Datalog clause set is eventually explored by the Datalog engine VLog [13]. This hammer constitutes a decision procedure for both universal and existential conjectures. The results of our experiments show that we can verify non-trivial existential and universal conjectures in the range of seconds while state-of-the-art solvers cannot solve all problems in reasonable time, see Section 4.

Related Work: Reasoning about $\mathrm{BS(LA)}$ clause sets is supported by SMT (Satisfiability Modulo Theories) [30,29]. In general, SMT comprises the combination of a number of theories beyond LA such as arrays, lists, strings, or bit vectors. While SMT is a decision procedure for the $\mathrm{BS(LA)}$ ground case, universally quantified variables can be considered by instantiation [36]. Reasoning by instantiation does result in a refutationally complete procedure for $\mathrm{BS(SLA)}$, but not in a decision procedure. The Horn fragment $\mathrm{HBS(LA)}$ out of $\mathrm{BS(LA)}$ is receiving additional attention [20,7], because it is well-suited for software analysis and verification. Research in this direction also goes beyond the theory of LA and considers minimal model semantics in addition, but is restricted to existential conjectures. Other research focuses on universal conjectures, but over non-arithmetic theories, e.g., invariant checking for array-based systems [14] or considers abstract decidability criteria incomparable with the $\mathrm{HBS(LA)}$ class [34]. Hierarchic superposition [3] and Simple Clause Learning over Theories (SCL(T)) [12] are both refutationally complete for $\mathrm{BS(LA)}$. While SCL(T) can be immediately turned into a decision procedure for even larger fragments than $\mathrm{BS(SLA)}$ [12], hierarchic superposition needs to be refined to become a decision procedure already because of the Bernays-Schönfinkel part [21]. Our Datalog hammer translates $\mathrm{HBS(SLA)}$ clause sets with both existential and universal conjectures into HBS clause sets which are also subject to first-order theorem proving. Instance generating approaches such as iProver [25] are a decision procedure for this fragment, whereas superposition-based [3] first-order provers such as E [38], SPASS [40], Vampire [37], have additional mechanisms implemented to decide HBS. In our experiments, Section 4, we will discuss the differences between all these approaches on a number of benchmark examples in more detail.

The paper is organized as follows: after a section on preliminaries, Section 2, we present the theory of our sorted Datalog hammer in Section 3, followed by experiments on real world supervisor verification conditions, Section 4. The paper ends with a discussion of the obtained results and directions for future work, Section 5. The artifact (including binaries of our tools and all benchmark problems) is available at [9]. An extended version is available at [10] including proofs and pseudo-code algorithms for the presented results.

2 Preliminaries

We briefly recall the basic logical formalisms and notations we build upon [11]. Starting point is a standard many-sorted first-order language for BS with *constants* (denoted a,b,c), without non-constant function symbols, *variables* (denoted w,x,y,z), and *predicates* (denoted P,Q,R) of some fixed *arity*. *Terms* (denoted t,s) are variables or constants. We write $\bar{x}$ for a vector of variables, $\bar{a}$ for a vector of constants, and so on. An *atom* (denoted A,B) is an expression $P(\bar{t})$ for a predicate P of arity n and a term list $\bar{t}$ of length n. A *positive literal* is an atom A and a *negative literal* is a negated atom $\neg A$. We define $\text{comp}(A) = \neg A$, $\text{comp}(\neg A) = A$, $|A| = A$ and $|\neg A| = A$. Literals are usually denoted L,K,H.

A *clause* is a disjunction of literals, where all variables are assumed to be universally quantified. C,D denote clauses, and N denotes a clause set. We write $\text{atoms}(X)$ for the set of atoms in a clause or clause set X. A clause is *Horn* if it contains at most one positive literal, and a *unit clause* if it has exactly one literal. A clause $A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$ can be written as an implication $B_1 \wedge \ldots \wedge B_m \rightarrow A_1 \vee \ldots \vee A_n$, still omitting universal quantifiers. If Y is a term, formula, or a set thereof, $\text{vars}(Y)$ denotes the set of all variables in Y, and Y is *ground* if $\text{vars}(Y) = \emptyset$. A *fact* is a ground unit clause with a positive literal.

Datalog and the Horn Bernays-Schönfinkel Fragment: The *Horn case of the Bernays-Schönfinkel fragment* (HBS) comprises all sets of clauses with at most one positive literal. The more general Bernays-Schönfinkel fragment (BS) in first-order logic allows arbitrary *formulas* over atoms, i.e., arbitrary Boolean connectives and leading existential quantifiers. BS formulas can be polynomially transformed into clause sets with common syntactic transformations while preserving satisfiability and all entailments that do not refer to auxiliary constants and predicates introduced in the transformation [32]. BS theories in our sense are also known as *disjunctive Datalog programs* [16], specifically when written as implications. A HBS clause set is also called a *Datalog program*. Datalog is sometimes viewed as a second-order language. We are only interested in query answering, which can equivalently be viewed as first-order entailment or second-order model checking [1]. Again, it is common to write clauses as implications in this case.

Two types of *conjectures*, i.e., formulas we want to prove as consequences of a clause set, are of particular interest: *universal* conjectures $\forall\bar{x}.\phi$ and *existential* conjectures $\exists\bar{x}.\phi$, where ϕ is a BS formula that only uses variables in $\bar{x}$. We call such a conjecture positive if the formula only uses conjunctions and disjunctions to connect atoms. Positive conjectures are the focus of our Datalog hammer and they have the useful property that they can be transformed to one atom over a fresh predicate symbol by adding some suitable Horn clause definitions to our clause set N [32,11]. This is also the reason why we assume for the rest of the paper that all relevant universal conjectures have the form $\forall\bar{x}.P(\bar{x})$ and existential conjectures the form $\exists\bar{x}.P(\bar{x})$.

A *substitution* σ is a function from variables to terms with a finite domain $\text{dom}(\sigma) = \{x \mid x\sigma \neq x\}$ and codomain $\text{codom}(\sigma) = \{x\sigma \mid x \in \text{dom}(\sigma)\}$. We denote substitutions by σ,δ,ρ. The application of substitutions is often written postfix, as in $x\sigma$, and is homomorphically extended to terms, atoms, literals, clauses, and quantifier-free formulas. A substitution σ is *ground* if $\text{codom}(\sigma)$ is ground. Let Y denote some term, literal, clause, or clause set. σ is a *grounding* for Y if $Y\sigma$ is ground, and $Y\sigma$ is a *ground instance* of Y in this case. We denote by $\text{gnd}(Y)$ the set of all ground instances of Y, and by $\text{gnd}_B(Y)$ the set of all ground instances over a given set of constants B. The *most general unifier* $\text{mgu}(Z_1,Z_2)$ of two terms/atoms/literals Z_1 and Z_2 is defined as usual, and we assume that it does not introduce fresh variables and is idempotent.

We assume a standard many-sorted first-order logic model theory, and write $\mathcal{A} \models \phi$ if an interpretation $\mathcal{A}$ satisfies a first-order formula ϕ. A formula ψ is a logical consequence of ϕ, written $\phi \models \psi$, if $\mathcal{A} \models \psi$ for all $\mathcal{A}$ such that $\mathcal{A} \models \phi$. Sets of clauses are semantically treated as conjunctions of clauses with all variables quantified universally.

BS **with Linear Arithmetic:** The extension of BS with linear arithmetic both over real and integer variables, BS(LA), is the basis for the formalisms studied in this paper. We extend the standard *many-sorted* first-order logic with finitely many first-order sorts $\mathcal{F}_i$ and with two arithmetic sorts $\mathcal{R}$ for the real numbers and $\mathcal{Z}$ for the integer numbers. The sort $\mathcal{Z}$ is a *subsort* of $\mathcal{R}$. Given a clause set N, the interpretations $\mathcal{A}$ of our sorts are fixed: $\mathcal{R}^{\mathcal{A}} = \mathbb{R}$, $\mathcal{Z}^{\mathcal{A}} = \mathbb{Z}$, and $\mathcal{F}_i^{\mathcal{A}} = \mathbb{F}_i$, i.e., a first-order sort interpretation $\mathbb{F}_i$ consists of the set of constants in N belonging to that sort, or a single constant out of the signature if no such constant occurs. Note that this is not a deviation from standard semantics in our context as for the arithmetic part the canonical domain is considered and for the first-order sorts BS has the finite model property over the occurring constants which is sufficent for refutation-based reasoning. This way first-order constants are distinct values.

Constant symbols, arithmetic function symbols, variables, and predicates are uniquely declared together with *sort* expressions. The unique sort of a constant symbol, variable, predicate, or term is denoted by the function $\mathrm{sort}(Y)$ and we assume all terms, atoms, and formulas to be well-sorted. The sort of predicate P's argument position i is denoted by $\mathrm{sort}(P,i)$. For arithmetic function symbols we consider the minimal sort with respect to the subsort relation between $\mathcal{R}$ and $\mathcal{Z}$. Eventually, we don't consider arithmetic functions here, so the subsort relationship boils down to substitute an integer sort variable or number for a real sort variable.

We assume *pure* input clause sets, which means the only constants of sort $\mathcal{R}$ or $\mathcal{Z}$ are numbers. This means the only constants that we do allow are integer numbers $c \in \mathbb{Z}$ and the constants defining our finite first-order sorts $\mathcal{F}_i$. Satisfiability of pure BS(LA) clause sets is semi-decidable, e.g., using *hierarchic superposition* [3] or *SCL(T)* [12]. Impure BS(LA) is no longer compact and satisfiability becomes undecidable, but it can be made decidable when restricting to ground clause sets [18].

All arithmetic predicates and functions are interpreted in the usual way. An interpretation of BS(LA) coincides with $\mathcal{A}^{\mathrm{LA}}$ on arithmetic predicates and functions, and freely interprets free predicates. For pure clause sets this is well-defined [3]. Logical satisfaction and entailment is defined as usual, and uses similar notation as for BS.

Example 1. The following BS(LA) clause from our ECU case study compares the values of engine speed (Rpm) and pressure (KPa) with entries in an ignition table (IgnTable) to derive the basis of the current ignition value (IgnDeg1):

$$\begin{aligned}&x_1 < 0 \lor x_1 \geq 13 \lor x_2 < 880 \lor x_2 \geq 1100 \lor \neg \mathrm{KPa}(x_3, x_1) \lor \\ &\neg \mathrm{Rpm}(x_4, x_2) \lor \neg \mathrm{IgnTable}(0, 13, 880, 1100, z) \lor \mathrm{IgnDeg1}(x_3, x_4, x_1, x_2, z)\end{aligned} \tag{1}$$

Terms of the two arithmetic sorts are constructed from a set $\mathcal{X}$ of *variables*, the set of integer constants $c \in \mathbb{Z}$, and binary function symbols $+$ and $-$ (written infix). Atoms in BS(LA) are either *first-order atoms* (e.g., $\mathrm{IgnTable}(0, 13, 880, 1100, z)$) or *(linear) arithmetic atoms* (e.g., $x_2 < 880$). Arithmetic atoms may use the predicates $\leq, <, \neq, =, >, \geq$, which are written infix and have the expected fixed interpretation. Predicates used in first-order atoms are called *free*. *First-order literals* and related notation is defined as before. *Arithmetic literals* coincide with arithmetic atoms, since the arithmetic predicates are closed under negation, e.g., $\neg(x_2 \geq 1100) \equiv x_2 < 1100$.

BS(LA) clauses and conjectures are defined as for BS but using BS(LA) atoms. We often write Horn clauses in the form $\Lambda \,\|\, \Delta \rightarrow H$ where Δ is a multiset of free first-order atoms, H is either a first-order atom or $\bot$, and Λ is a multiset of LA atoms. The semantics of a clause in the form $\Lambda \,\|\, \Delta \to H$ is $\bigvee_{\lambda\in\Lambda} \neg\lambda \vee \bigvee_{A\in\Delta} \neg A \vee H$, e.g., the clause $x > 1 \vee y \neq 5 \vee \neg Q(x) \vee R(x,y)$ is also written $x \leq 1, y = 5 \| Q(x) \to R(x,y)$.

A clause or clause set is *abstracted* if its first-order literals contain only variables or first-order constants. Every clause C is equivalent to an abstracted clause that is obtained by replacing each non-variable arithmetic term t that occurs in a first-order atom by a fresh variable x while adding an arithmetic atom $x \neq t$ to C. We asssume abstracted clauses for theory development, but we prefer non-abstracted clauses in examples for readability,e.g., a fact $P(3,5)$ is considered in the development of the theory as the clause $x = 3, x = 5 \,\|\!\to P(x,y)$, this is important when collecting the necessary test points. Moreover, we assume that all variables in the theory part of a clause also appear in the first order part, i.e., $\text{vars}(\Lambda) \subseteq \text{vars}(\Delta \to H)$ for every clause $\Lambda \,\|\, \Delta \to H$. If this is not the case for x in $\Lambda \,\|\, \Delta \to H$, then we can easily fix this by first introducing a fresh unary predicate Q over the $\text{sort}(x)$, then adding the literal $Q(x)$ to Δ, and finally adding a clause $\| \to Q(x)$ to our clause set. Alternatively, x could be eliminated by LA variable elimintation in our context, however this results in a worst case exponential blow up in size. This restriction is necessary because we base all our computations for the test-point scheme on predicate argument positions and would not get any test points for variables that are not connected to any predicate argument positions.

Simpler Forms of Linear Arithmetic: The main logic studied in this paper is obtained by restricting HBS(LA) to a simpler form of linear arithmetic. We first introduce a simpler logic HBS(SLA) as a well-known fragment of HBS(LA) for which satisfiability is decidable [19,22], and later present the generalization HBS(LA)PA of this formalism that we will use.

Definition 2. *The* Horn Bernays-Schönfinkel fragment over simple linear arithmetic, HBS(SLA), *is a subset of* HBS(LA) *where all arithmetic atoms are of the form $x \triangleleft c$ or $d \triangleleft c$, such that $c \in \mathbb{Z}$, d is a (possibly free) constant, $x \in \mathcal{X}$, and $\triangleleft \in \{\leq, <, \neq, =, >, \geq\}$.*

Please note that HBS(SLA) clause sets may be unpure due to free first-order constants of an arithmetic sort. Studying unpure fragments is beyond the scope of this paper but they show up in applications as well.

Example 3. The ECU use case leads to HBS(LA) clauses such as

$$\begin{aligned} &x_1 < y_1 \vee x_1 \geq y_2 \vee x_2 < y_3 \vee x_2 \geq y_4 \vee \neg\text{KPa}(x_3, x_1) \vee \\ &\neg\text{Rpm}(x_4, x_2) \vee \neg\text{IgnTable}(y_1, y_2, y_3, y_4, z) \vee \text{IgnDeg1}(x_3, x_4, x_1, x_2, z). \end{aligned} \tag{2}$$

This clause is not in HBS(SLA), e.g., since $x_1 > x_5$ is not allowed in BS(SLA). However, clause (1) of Example 1 is a BS(SLA) clause that is an instance of (2), obtained by the substitution $\{y_1 \mapsto 0, y_2 \mapsto 13, y_3 \mapsto 880, y_4 \mapsto 1100\}$. This grounding will eventually be obtained by resolution on the IgnTable predicate, because it occurs only positively in ground unit facts.

Example 3 shows that HBS(SLA) clauses can sometimes be obtained by instantiation. In fact, for the satisfiability of an HBS(LA) clause set N only those instances of clauses $(\Lambda \,\|\, \Delta \to H)\sigma$ are *relevant*, for which we can actually derive all ground facts $A \in \Delta\sigma$ by resolution from N. If A cannot be derived from N and N is satisfiable, then there always exists a satisfying interpretation $\mathcal{A}$ that interprets A as false (and thus $(\Lambda \,\|\, \Delta \to H)\sigma$ as true). Moreover, if those relevant instances can be simplified to HBS(SLA) clauses, then it is possible to extend almost all HBS(SLA) techniques (including our Datalog hammer) to those HBS(LA) clause sets.

In our case resolution means *hierarchic unit resolution*: given a clause $\Lambda_1 \,\|\, L, \Delta \rightarrow H$ and a unit clause $\Lambda_2 \,\|\!\rightarrow K$ with $\sigma = \mathrm{mgu}(L,K)$, their *hierarchic resolvent* is $(\Lambda_1, \Lambda_2 \,\|\, \Delta \rightarrow H)\sigma$. A fact $P(\bar{a})$ is *derivable* from a pure set of HBS(LA) clauses N if there exists a clause $\Lambda \,\|\!\rightarrow P(\bar{t})$ that (i) is the result of a sequence of unit resolution steps from the clauses in N and (ii) has a grounding σ such that $P(\bar{t})\sigma = P(\bar{a})$ and $\Lambda\sigma$ evaluates to true. If N is satisfiable, then this means that any fact $P(\bar{a})$ derivable from N is true in all satisfiable interpretations of N, i.e., $N \models P(\bar{a})$. We denote the *set of derivable facts* for a predicate P from N by $\mathrm{dfacts}(P,N)$. A *refutation* is the sequence of resolution steps that produces a clause $\Lambda \,\|\!\rightarrow \bot$ with $\mathcal{A}^{\mathrm{LA}} \models \Lambda\delta$ for some grounding δ. *Hierarchic unit resolution* is sound and refutationally complete for pure HBS(LA), since every set N of pure HBS(LA) clauses N is *sufficiently complete* [3], and hence *hierarchic superposition* is sound and refutationally complete for N [3,6].

So naturally if all derivable facts of a predicate P already appear in N, then only those instances of clauses can be relevant whose occurrences of P match those facts (i.e., can be resolved with them). We call predicates with this property positively grounded:

Definition 4 (Positively Grounded Predicate [11]). *Let N be a set of* HBS(LA) *clauses. A free first-order predicate P is a* positively grounded predicate *in N if all positive occurrences of P in N are in ground unit clauses (also called facts).*

Definition 5 (Positively Grounded HBS(SLA): HBS(SLA)P [11]). *An* HBS(*LA*) *clause set N is out of the fragment* positively grounded HBS(SLA) (HBS(SLA)P) *if we can transform N into an* HBS(SLA) *clause set N' by first resolving away all negative occurrences of positively grounded predicates P in N, simplifying the thus instantiated* LA *atoms, and finally eliminating all clauses where those predicates occur negatively.*

As mentioned before, if all relevant instances of an HBS(LA) clause set can be simplified to HBS(SLA) clauses, then it is possible to extend almost all HBS(SLA) techniques (including our Datalog hammer) to those clause sets. HBS(SLA)P clause sets have this property and this is the reason, why we managed to extend our Datalog hammer to pure HBS(SLA)P clause sets in [11]. For instance, the set $N = \{P(1), P(2), Q(0), (x \leq y + z \,\|\, P(y), Q(z) \rightarrow R(x,y))\}$ is an HBS(LA) clause set, but not an HBS(SLA) clause set due to the inequality $x \leq y + z$. Note, however, that the predicates P and Q are positively grounded, the only positive occurrences of P and Q are the facts $P(1)$, $P(2)$, and $Q(0)$. If we resolve with the facts for P and Q and simplify, then we get the clause set $N' = \{P(1), P(2), Q(0), (x \leq 1 \,\|\!\rightarrow R(x,1)), (x \leq 2 \,\|\!\rightarrow R(x,2))\}$, which does now belong to HBS(SLA). This means N is a positively grounded HBS(SLA) clause set and our Datalog hammer can still handle it.

Positively grounded predicates are only one way to filter out irrelevant clause instances. As part of our improvements, we define in Section 3 a new logic called approximately grounded HBS(SLA) (HBS(SLA)PA) that is an extension of HBS(SLA)P and serves as the new input logic of our sorted Datalog hammer.

Test-Point Schemes and Functions The Datalog hammer in [11] is based on the following idea: For any pure HBS(SLA) clause set N that is unsatisfiable, we only need to look at the instances $\mathrm{gnd}_B(N)$ of N over finitely many test points B to construct a refutation. Symmetrically, if N is satisfiable, then we can extrapolate a satisfying interpretation for N from a satisfying interpretation for $\mathrm{gnd}_B(N)$. If we can compute such a set of test points B for a clause set N, then we can transform the clause set into an equisatisfiable Datalog program. There exist similar properties for universal/existential conjectures. A *test-point scheme* is an algorithm that can compute such a set of test points B for any HBS(SLA) clause set N and any conjecture $N \models Q\bar{x}.P(\bar{x})$ with $Q \in \{\exists, \forall\}$.

The test-point scheme used by our original Datalog hammer computes the same set of test points for all variables and predicate argument positions. This has several disadvantages: (i) it cannot handle variables with different sorts and (ii) it often selects too many test points (per argument position) because it cannot recognize which inequalities and which argument positions are connected. The goal of this paper is to resolve these issues. However, this also means that we have to assign different test-point sets to different predicate argument positions. We do this with so-called test-point functions.

A *test-point function* (tp-function) β is a function that assigns to some argument positions i of some predicates P a set of test points $\beta(P,i)$. An argument position (P,i) is assigned a set of test points if $\beta(P,i) \subseteq \mathrm{sort}(P,i)^{\mathcal{A}}$ and otherwise $\beta(P,i) = \bot$. A test-point function β is *total* if all argument positions (P,i) are assigned, i.e., $\beta(P,i) \neq \bot$.

A variable x of a clause $\Lambda \parallel \Delta \to H$ occurs in an argument position (P,i) if $(P,i) \in \mathrm{depend}(x, \Lambda \parallel \Delta \to H)$, where $\mathrm{depend}(x,Y) = \{(P,i) \mid P(\bar{t}) \in \mathrm{atoms}(Y) \text{ and } t_i = x\}$. Similarly, a variable x of an atom $Q(\bar{t})$ occurs in an argument position (Q,i) if $(Q,i) \in \mathrm{depend}(x, Q(\bar{t}))$. A substitution σ for a clause Y or atom Y is a *well-typed instance* over a tp-function β if it guarantees for each variable x that $x\sigma$ is an element of $\mathrm{sort}(x)^{\mathcal{A}}$ and part of every test-point set (i.e., $x\sigma \in \beta(P,i)$) of every argument position (P,i) it occurs in (i.e., $(P,i) \in \mathrm{depend}(x,Y)$) and that is assigned a test-point set by β (i.e., $\beta(P,i) \neq \bot$). To abbreviate this, we define a set $\mathrm{wti}(x,Y,\beta)$ that contains all values with which a variable can fulfill the above condition, i.e., $\mathrm{wti}(x,Y,\beta) = \mathrm{sort}(x)^{\mathcal{A}} \cap (\bigcap_{(P,i) \in \mathrm{depend}(x,Y) \text{ and } \beta(P,i) \neq \bot} \beta(P,i))$. Following this definition, we denote by $\mathrm{wtis}_{\beta}(Y)$ the *set of all well-typed instances* for a clause/atom Y over the tp-function β, or formally: $\mathrm{wtis}_{\beta}(Y) = \{\sigma \mid \forall x \in \mathrm{vars}(Y).(x\sigma) \in \mathrm{wti}(x,Y,\beta)\}$. With the function gnd_{β}, we denote the *set of all well-typed ground instances* of a clause/atom Y over the tp-function β, i.e., $\mathrm{gnd}_{\beta}(Y) = \{Y\sigma \mid \sigma \in \mathrm{wtis}_{\beta}(Y)\}$, or a set of clauses N, i.e., $\mathrm{gnd}_{\beta}(N) = \{Y\sigma \mid Y \in N \text{ and } \sigma \in \mathrm{wtis}_{\beta}(Y)\}$.

The most general tp-function, denoted by β^*, assigns each argument position to the interpretation of its sort, i.e., $\beta^*(P,i) = \mathrm{sort}(P,i)^{\mathcal{A}}$. So depending on the sort of (P,i), either to $\mathbb{R}$, $\mathbb{Z}$, or one of the $\mathbb{F}_i$. A set of clauses N is satisfiable if and only if $\mathrm{gnd}_{\beta^*}(N)$, the set of all ground instances of N over the base sorts, is satisfiable. Since β^* is the most general tp-function, we also write $\mathrm{gnd}(Y)$ for $\mathrm{gnd}_{\beta^*}(Y)$ and $\mathrm{wtis}(Y)$ for $\mathrm{wtis}_{\beta^*}(Y)$.

If we restrict ourselves to test points, then we also only get interpretations over test points and not for the full base sorts. In order to extrapolate an interpretation from test points to their full sorts, we define extrapolation functions (ep-functions) η. An *extrapolation function* (ep-function) $\eta(P,\bar{a})$ maps an argument vector of test points for predicate P (with $a_i \in \beta(P,i)$) to the subset of $\mathrm{sort}(P,1)^{\mathcal{A}} \times \ldots \times \mathrm{sort}(P,n)^{\mathcal{A}}$ that is supposed to be interpreted the same as $\bar{a}$, i.e., $P(\bar{a})$ is interpreted as true if and only if $P(\bar{b})$ with $\bar{b} \in \eta(P,\bar{a})$ is interpreted as true. By default, any argument vector of test points $\bar{a}$ for P must also be an element of $\eta(P,\bar{a})$, i.e., $\bar{a} \in \eta(P,\bar{a})$. An extrapolation function does not have to be complete for all argument positions, i.e., there may exist argument positions from which we cannot extrapolate to all argument vectors. Formally this means that the actual set of values that can be extrapolated from (P,i) (i.e., $\bigcup_{a_1 \in \beta(P,1)} \ldots \bigcup_{a_n \in \beta(P,n)} \eta(P,\bar{a})$) may be a strict subset of $\mathrm{sort}(P,1)^{\mathcal{A}} \times \ldots \times \mathrm{sort}(P,n)^{\mathcal{A}}$. For all other values $\bar{a}$, $P(\bar{a})$ is supposed to be interpreted as false.

Covering Clause Sets and Conjectures Our goal is to create total tp-functions that restrict our solution space from the infinite reals and integers to finite sets of test points while still preserving (un)satisfiability. Based on these tp-functions, we are then able to define a Datalog hammer that transforms a clause set belonging to (an extension of) HBS(LA) into an equisatisfiable HBS clause set; even modulo universal and existential conjectures.

To be more precise, we are interested in finite tp-functions (together with matching ep-functions) that cover a clause set N or a conjecture $N \models Q\bar{x}.P(\bar{x})$ with $Q \in \{\exists, \forall\}$. A total tp-function β is *finite* if each argument position is assigned to a finite set of test points, i.e., $|\beta(P,i)| \in \mathbb{N}$. A tp-function β *covers a set of clauses* N if $\mathrm{gnd}_\beta(N)$ is equisatisfiable to N. A tp-function β *covers a universal conjecture* $\forall\bar{x}.Q(\bar{x})$ over N if $\mathrm{gnd}_\beta(N) \cup N_Q$ is satisfiable if and only if $N \models \forall\bar{x}.Q(\bar{x})$ is false. Here N_Q is the set $\{\| \mathrm{gnd}_\beta(Q(\bar{x})) \rightarrow \bot\}$ if η is complete for Q or the empty set otherwise. A tp-function β *covers an existential conjecture* $N \models \exists\bar{x}.Q(\bar{x})$ if $\mathrm{gnd}_\beta(N) \cup \mathrm{gnd}_\beta(\| Q(\bar{x}) \rightarrow \bot)$ is satisfiable if and only if $N \models \exists\bar{x}.Q(\bar{x})$ is false.

The most general tp-function β^* obviously covers all HBS(LA) clause sets and conjectures because satisfiability of N is defined over $\mathrm{gnd}_{\beta^*}(N)$. However, β^* is not finite. The test-point scheme in [11], which assigns one finite set of test points B to all variables, also covers clause sets and universal/existential conjectures; at least if we restrict our input to variables over the reals. As mentioned before, the goal of this paper is improve this test-point scheme by assigning different test-point sets to different predicate argument positions.

3 The Sorted Datalog Hammer

In this section, we present a transformation that we call the *sorted Datalog hammer*. It transforms any pure HBS(SLA) clause set modulo a conjecture into an HBS clause set. To guide our explanations, we apply each step of the transformation to a simplified example of the electronic control unit use case:

Example 6. An electronic control unit (ECU) of a combustion engine determines actuator operations. For instance, it computes the ignition timings based on a set of input sensors. To this end, it looks up some base factors from static tables and combines them to the actual actuator values through a series of rules.

In our simplified model of an ECU, we only compute one actuator value, the ignition timing, and we only have an engine speed sensor (measuring in Rpm) as our input sensor. Our verification goal, expressed as a universal conjecture, is to confirm, that the ECU computes an ignition timing for all potential input sensor values. Determining completeness of a set of rules, i.e., determining that the rules produce a result for all potential input values, is also our most common application for universal conjectures. The ECU model is encoded as the following pure HBS(LA) clause set N:
D_1: SpeedTable(0,2000,1350), D_2: SpeedTable(2000,4000,1600),
D_3: SpeedTable(4000,6000,1850), D_4: SpeedTable(6000,8000,2100),
$C_1: 0 \leq x_p, x_p < 8000 \| \rightarrow \mathrm{Speed}(x_p)$,
$C_2: x_1 \leq x_p, x_p < x_2 \| \mathrm{Speed}(x_p), \mathrm{SpeedTable}(x_1,x_2,y) \rightarrow \mathrm{IgnDeg}(x_p,y)$,
$C_3: \mathrm{IgnDeg}(x_p,z) \rightarrow \mathrm{ResArgs}(x_p)$, $C_4: \mathrm{ResArgs}(x_p) \rightarrow \mathrm{Conj}(x_p)$,
$C_5: x_p \geq 8000 \| \rightarrow \mathrm{Conj}(x_p)$, $C_6: x_p < 0 \| \rightarrow \mathrm{Conj}(x_p)$,

In this example all variables are real variables. The clauses $D_1 - D_4$ are table entries from which we determine the base factor of our ignition time based on the speed. Semantically, D_1 : SpeedTable(0,2000,1350) states that the base ignition time is 13.5° before dead center if the engine speed lies between 0Rpm and 2000Rpm. The clause C_1 produces all possible input sensor values labeled by the predicate Speed. The clause C_2 determines the ignition timing from the current speed and the table entries. The end result is stored in the predicate $\mathrm{IgnDeg}(x_p,z)$, where z is the resulting ignition timing and x_p is the speed that led to this result. The clauses $C_3 - C_6$ are necessary for encoding the verification goal as a universal conjecture over a single atom. In clause C_3, the return value is removed from the result predicate $\mathrm{IgnDeg}(x_p,z)$ because for the conjecture we only need to know that there is a result and not what the result is. Clause C_4 guarantees that the conjecture predicate $\mathrm{Conj}(x_p)$ is true if the rules can produce a $\mathrm{IgnDeg}(x_p,z)$ for the sensor value. Clauses C_5&C_6 guarantee that the conjecture predicate is true if one of the sensor values is out of bounds. This flattening process can be done automatically using the techniques outlined in [11]. Hence, the ECU computes an ignition timing for all potential input sensor values if the universal conjecture $\forall x_p.\mathrm{Conj}(x_p)$ is entailed by N.

Approximately Grounded Example 6 contains inequalities that go beyond simple variable bounds, e.g., $x_1 \leq x_p$ in C_2. However, it is possible to reduce the example to an HBS(SLA) clause set. As our first step of the sorted Datalog hammer, we explain a way to heuristically determine which HBS(LA) clause sets can be reduced to HBS(SLA) clause sets. Moreover, we show later that we do not have to explicitly perform this reduction but that we can extend our other algorithms to handle this heuristic extension of HBS(SLA) directly.

We start by formulating an extension of positively grounded HBS(SLA) called approximately grounded HBS(SLA). It is based on over-approximating the set of *derivable values* $\mathrm{dvals}(P,i,N) = \{a_i \mid P(\bar{a}) \in \mathrm{dfacts}(P,N)\}$ for each argument position i of each predicate P in N with only finitely many derivable values, i.e., $|\mathrm{dvals}(P,i,N)| \in \mathbb{N}$. These argument positions are also called *finite*. Naturally, all argument positions over first-order sorts $\mathcal{F}$ are finite argument positions. With regard to clause relevance, only those clause instances are relevant, where a finite argument position is instantiated by one of the derivable values. We call a set of clauses N an approximately grounded HBS(SLA) clause set if all relevant instances based on this criterion can be simplified to HBS(SLA) clauses. For instance, the set $N = \{(x \leq 1 \,\|\, \rightarrow P(x,1)), (x > 2 \,\|\, \rightarrow P(x,3)), (x \geq 0 \,\|\, \rightarrow Q(x,0)), (u \leq y+z \,\|\, P(x,y), Q(x,z) \rightarrow R(x,y,z,u))\}$ is an HBS(LA) clause set, but not a (positively grounded) HBS(SLA) clause set due to the inequality $z \leq y+u$ and the lack of positively grounded predicates. However, the argument positions $(P,2)$, $(Q,2)$, $(R,2)$ and $(R,3)$ only have finitely many derivable values $\mathrm{dvals}(P,2,N) = \mathrm{dvals}(R,2,N) = \{1,3\}$ and $\mathrm{dvals}(Q,2,N) = \mathrm{dvals}(R,3,N) = \{0\}$. If we instantiate all occurrences of P and Q over those values, then we get the set $N' = \{(x \leq 1 \,\|\, \rightarrow P(x,1)), (x > 2 \,\|\, \rightarrow P(x,3)), (x \geq 0 \,\|\, \rightarrow Q(x,0)), (u \leq 1 \,\|\, P(x,1), Q(x,0) \rightarrow R(x,1,0,u)), (u \leq 3 \,\|\, P(x,3), Q(x,0) \rightarrow R(x,3,0,u))\}$ that is an HBS(SLA) clause set. This means N is an approximately grounded HBS(SLA) clause set and our extended Datalog hammer can handle it.

Determining the finiteness of a predicate argument position (and all its derivable values) is not trivial. In general, it is as hard as determining the satisfiability of a clause set [10], so in the case of HBS(LA) undecidable [15,23]. This is the reason, why we only over-approximate the derivable values with the following algorithm.

```
DeriveValues$(N)$
for all predicates $P$ and argument positions $i$ for $P$
    $\mathrm{avals}(P,i,N) := \emptyset$;
change $:= \top$;
while (change)
    change $:= \bot$;
    for all Horn clauses $\Lambda \,\|\, \Delta \rightarrow P(t_1,...,t_n) \in N$
        for all argument positions $1 \leq i \leq n$ where $\mathrm{avals}(P,i,N) \neq \mathbb{R}$
            if $[(t_i = c)$ or $t_i$ is assigned a constant $c$ in $\Lambda$ and $c \notin \mathrm{avals}(P,i,N)]$ then
                $\mathrm{avals}(P,i,N) := \mathrm{avals}(P,i,N) \cup \{c\}$,change $:= \top$;
            else if $[t_i$ appears in argument positions $(Q_1,k_1),...,(Q_m,k_m)$ in $\Delta$
                    and $\mathrm{avals}(P,i,N) \not\supseteq \bigcap_j \mathrm{avals}(Q_j,k_j,N)$ ] then
                if $[\mathbb{R} \neq \bigcap_j \mathrm{avals}(Q_j,k_j,N)]$ then
                    $\mathrm{avals}(P,i,N) := \mathrm{avals}(P,i,N) \cup \bigcap_j \mathrm{avals}(Q_j,k_j,N)$,change $:= \top$;
                else
                    $\mathrm{avals}(P,i,N) := \mathbb{R}$,change $:= \top$;
```

At the start, DeriveValues(N) sets $\mathrm{avals}(P,i,N) = \emptyset$ for all predicate argument positions. Then it repeats iterating over the clauses in N and uses the current sets avals in order to derive new values, until it reaches a fixpoint. Whenever, DeriveValues(N) computes that a clause can derive infinitely many values for an argument position, it simply sets $\mathrm{avals}(P,i,N) = \mathbb{R}$ for both real and integer argument positions. This is the case, when we have a clause $\Lambda \,\|\, \Delta \rightarrow P(t_1,...,t_n)$, and an argument position i for P, such that: (i) t_i is not a constant (and therefore a variable), (ii) t_i is not assigned a constant c in Λ (i.e., there is no equation $t_i = c$ in Λ), (iii) t_i is only connected to argument positions $(Q_1,k_1),...,(Q_m,k_m)$ in Δ that already have $\mathrm{avals}(Q_j,k_j,N) = \mathbb{R}$. The latter also includes the case that t_i is not connected to any argument positions in Δ. For instance, DeriveValues(N) would recognize that clause C_1 in example 6 can be used to derive infinitely many values for the argument position (Speed, 1) because the variable x_p is not assigned an equation in C_1's theory constraint $\Lambda := (0 \leq x_p, x_p < 8000)$ and x_p is not connected to any argument position on the left side of the implication. Hence, DeriveValues(N) would set $\mathrm{avals}(\mathrm{Speed},1,N) = \mathbb{R}$.

For each run through the while loop, at least one predicate argument position is set to $\mathbb{R}$ or the set is extended by at least one constant. The set of constants in N as well as the number of predicate argument positions in N are finite, hence DeriveValues(N) terminates. It is correct because in each step it over-approximates the result of a hierarchic unit resulting resolution step, see Section 2. The above algorithm is highly inefficient. In our own implementation, we only apply it if all clauses are non-recursive and by first ordering the clauses based on their dependencies. This guarantees that every clause is visited at most once and is sufficient for both of our use cases.

Based on avals, we can now build a tp-function β^a that maps all finite argument positions (P,i) that our over-approximation detected to the over-approximation of their derivable values, i.e., $\beta^a(P,i) := \mathrm{avals}(P,i,N)$ if $|\mathrm{avals}(P,i,N)| \in \mathbb{N}$ and $\beta^a(P,i) := \bot$ otherwise. With β^a we derive the finitely grounded over-approximation $\mathrm{agnd}(Y)$ of a set of clauses Y, a clause Y or an atom Y. This set is equivalent to $\mathrm{gnd}_{\beta^a}(Y)$, except that we assume that all LA atoms are simplified until they contain at most one integer number and that LA atoms that can be evaluated are reduced to true and false and the respective clause simplified. Based of $\mathrm{agnd}(N)$ we define a new extension of HBS(SLA) called approximately grounded HBS(SLA):

Definition 7 (Approximately Grounded $\mathrm{HBS(SLA)}$**:** $\mathrm{HBS(SLA)A}$**).** *A clause set N is out of the fragment* approximately grounded $\mathrm{HBS(SLA)}$ *or short* $\mathrm{HBS(SLA)A}$ *if* $\mathrm{agnd}(N)$ *is out of the* $\mathrm{HBS(SLA)}$ *fragment. It is called* $\mathrm{HBS(SLA)PA}$ *if it is also pure.*

Example 8. Executing $\mathrm{DeriveValues}(N)$ on example 6 leads to the following results:
$\mathrm{avals}(\mathrm{SpeedTable},1,N)=\{0,2000,4000,6000\}$,
$\mathrm{avals}(\mathrm{SpeedTable},2,N)=\{2000,4000,6000,8000\}$,
$\mathrm{avals}(\mathrm{SpeedTable},3,N)=\{1350,1600,1850,2100\}$,
$\mathrm{avals}(\mathrm{IgnDeg},2,N)=\{1350,1600,1850,2100\}$,
and all other argument positions (P,i) are infinite so $\mathrm{avals}(P,i,N)=\mathbb{R}$ for them.

We can now easily check whether $\mathrm{agnd}(N)$ would turn our clause set into an $\mathrm{HBS(SLA)}$ fragment by checking whether the following holds for all inequalities: all variables in the inequality except for one must be connected to a finite argument position on the left side of the clause it appears in. This guarantees that all but one variable will be instantiated in $\mathrm{agnd}(N)$ and the inequality can therefore be simplified to a variable bound.

Connecting Argument Positions and Selecting Test Points As our second step, we are reducing the number of test points per predicate argument position by incorporating that not all argument positions are connected to all inequalities. This also means that we select different sets of test points for different argument positions. For finite argument positions, we can simply pick $\mathrm{avals}(P,i,N)$ as its set of test points. However, before we can compute the test-point sets for all other argument positions, we first have to determine to which inequalities and other argument positions they are connected.

Let N be an $\mathrm{HBS(SLA)PA}$ clause set and (P,i) an argument position for a predicate in N. Then we denote by $\mathrm{conArgs}(P,i,N)$ the *set of connected argument positions* and by $\mathrm{conIneqs}(P,i,N)$ the *set of connected inequalities*. Formally, $\mathrm{conArgs}(P,i,N)$ is defined as the minimal set that fulfills the following conditions: (i) two argument positions (P,i) and (Q,j) are connected if they share a variable in a clause in N, i.e., $(Q,j)\in\mathrm{conArgs}(P,i,N)$ if $(\Lambda \parallel \Delta \rightarrow H)\in N$, $P(\bar{t}),Q(\bar{s})\in\mathrm{atoms}(\Delta\cup\{H\})$, and $t_i=s_j=x$; and (ii) the connection relation is transitive, i.e., if $(Q,j)\in\mathrm{conArgs}(P,i,N)$, then $\mathrm{conArgs}(P,i,N)=\mathrm{conArgs}(Q,j,N)$. Similarly, $\mathrm{conIneqs}(P,i,N)$ is defined as the minimal set that fulfills the following conditions: (i) an argument position (P,i) is connected to an instance λ' of an inequality λ if they share a variable in a clause in N, i.e., $\lambda'\in\mathrm{conIneqs}(P,i,N)$ if $(\Lambda \parallel \Delta \rightarrow H)\in N$, $P(\bar{t})\in\mathrm{atoms}(\Delta\cup\{H\})$, $t_i=x$, $(\Lambda' \parallel \Delta' \rightarrow H')\in\mathrm{agnd}(\Lambda \parallel \Delta \rightarrow H)$, $\lambda'\in\Lambda'$, and $\lambda'=x\triangleleft c$ (where $\triangleleft=\{<,>,\leq,\geq,=,\neq\}$ and $c\in\mathbb{Z}$); (ii) an argument position (P,i) is connected to a value $c\in\mathbb{Z}$ if $P(\bar{t})$ with $t_i=c$ appears in a clause in N, i.e., $(x=c)\in\mathrm{conIneqs}(P,i,N)$ if $(\Lambda \parallel \Delta \rightarrow H)\in N$, $P(\bar{t})\in\mathrm{atoms}(\Delta\cup\{H\})$, and $t_i=c$; (iii) an argument position (P,i) is connected to a value $c\in\mathbb{Z}$ if (P,i) is finite and $c\in\mathrm{avals}(P,i,N)$, i.e., $(x=c)\in\mathrm{conIneqs}(P,i,N)$ if (P,i) is finite and $c\in\mathrm{avals}(P,i,N)$; and (iv) the connection relation is transitive, i.e., $\lambda\in\mathrm{conArgs}(Q,j,N)$ if $\lambda\in\mathrm{conIneqs}(P,i,N)$ and $(Q,j)\in\mathrm{conArgs}(P,i,N)$.

Example 9. To highlight the connections in example 6 more clearly, we use the same variable symbol for connected argument positions. Therefore $(\mathrm{SpeedTable},1)$ and $(\mathrm{SpeedTable},2)$ are only connected to themselves and $\mathrm{conArgs}(\mathrm{SpeedTable},3,N)=\{(\mathrm{SpeedTable},3),(\mathrm{IgnDeg},2)\}$, and $\mathrm{conArgs}(\mathrm{Speed},1,N)=\{(\mathrm{Speed},1),(\mathrm{IgnDeg},1),(\mathrm{ResArgs},1),(\mathrm{Conj},1)\}$, Computing the connected argument positions is a little bit more complicated: first, if a connected argument position is finite, then we have to add all values in avals as equations to the connected inequalities. E.g., $\text{conIneqs}(\text{SpeedTable},1,N) = \{x_1 = 0, x_1 = 2000, x_1 = 4000, x_1 = 6000\}$ because $\text{avals}(\text{SpeedTable},1,N) = \{0,2000,4000,6000\}$. Second, we have to add all inequalities connected in $\text{agnd}(N)$. Again this is possible without explicitly computing $\text{agnd}(N)$. E.g., for the inequality $x_1 \leq x_p$ in clause C_2, we determine that x_1 is connected to the finite argument position $(\text{SpeedTable},1)$ in C_2 and x_p is not connected to any finite argument positions. Hence, we have to connect the following variable bounds to all argument positions connected to x_p, i.e., $\{x_1 \leq x_p \mid x_1 \in \text{avals}(\text{SpeedTable},1,N)\} = \{x_p \geq 0, x_p \geq 2000, x_p \geq 4000, x_p \geq 6000\}$ to the argument positions $\text{conArgs}(\text{Speed},1,N)$. If we apply the above two steps to all clauses, then we get as connected inequalities: $\text{conIneqs}(\text{SpeedTable},2,N) = \{x_2 = 2000, x_2 = 4000, x_3 = 6000, x_4 = 8000\}$, $\text{conIneqs}(\text{SpeedTable},3,N) = \{y = 1350, y = 1600, y = 1850, y = 2100\}$, and $\text{conIneqs}(\text{Speed},1,N) = \{x_p < 0, x_p < 2000, x_p < 4000, x_p < 6000, x_p < 8000, x_p \geq 0, x_p \geq 2000, x_p \geq 4000, x_p \geq 6000, x_p \geq 8000\}$.

Now based on these sets we can construct a set of test points as follows: For each argument position (P,i), we partition the reals $\mathbb{R}$ into intervals such that any variable bound in $\lambda \in \text{conIneqs}(P,i,N)$ is satisfied by all points in one such interval I or none. Since we are in the Horn case, this is enough to ensure that we derive facts *uniformly* over those intervals and the integers/non-integers. To be more precise, we derive facts *uniformly* over those intervals and the integers because $P(\bar{a})$ is derivable from N and $a_i \in I \cap \mathbb{Z}$ implies that $P(\bar{b})$ is also derivable from N, where $b_j = a_j$ for $i \neq j$ and $b_i \in I \cap \mathbb{Z}$. Similarly, we derive facts *uniformly* over those intervals and the non-integers because $P(\bar{a})$ is derivable from N and $a_i \in I \setminus \mathbb{Z}$ implies that $P(\bar{b})$ is also derivable from N, where $b_j = a_j$ for $i \neq j$ and $b_i \in I$. As a result, it is enough to pick (if possible) one integer and one non-integer test point per interval to cover the whole clause set.

Formally we compute the interval partition $\text{iPart}(P,i,N)$ and the set of test points $\text{tps}(P,i,N)$ as follows: First we transform all variable bounds $\lambda \in \text{conIneqs}(P,i,N)$ into interval borders. A variable bound $x \lhd c$ with $\lhd \in \{\leq,<,>,\geq\}$ in $\text{conIneqs}(P,i,N)$ is turned into two interval borders. One of them is the interval border implied by the bound itself and the other its negation, e.g., $x \geq 5$ results in the interval border $[5$ and the interval border of the negation $5)$. Likewise, we turn every variable bound $x \lhd c$ with $\lhd \in \{=,\neq\}$ into all four possible interval borders for c, i.e. $c)$, $[c$, $c]$, and $(c$. The set of interval borders $\text{iEP}(P,i,N)$ is then defined as follows:

$$\begin{aligned}\text{iEP}(P,i,N) = \;& \{c],(c \mid x \lhd c \in \text{conIneqs}(P,i,N) \text{ where } \lhd \in \{\leq,=,\neq,>\}\} \cup \\ & \{c),[c \mid x \lhd c \in \text{conIneqs}(P,i,N) \text{ where } \lhd \in \{\geq,=,\neq,<\}\} \cup \{(-\infty,\infty)\}\end{aligned}$$

The interval partition $\text{iPart}(P,i,N)$ can be constructed by sorting $\text{iEP}(P,i,N)$ in an ascending order such that we first order by the border value—i.e. $\delta < \epsilon$ if $\delta \in \{c),[c,c],(c\}$, $\epsilon \in \{d),[d,d],(d\}$, and $c < d$—and then by the border type—i.e. $c) < [c < c] < (c$. The result is a sequence $[\ldots,\delta_l,\delta_u,\ldots]$, where we always have one lower border δ_l, followed by one upper border δ_u. We can guarantee that an upper border δ_u follows a lower border δ_l because $\text{iEP}(P,i,N)$ always contains $c)$ together with $[c$ and $c]$ together with $(c$ for $c \in \mathbb{Z}$, so always two consecutive upper and lower borders. Together with $(-\infty$ and $\infty)$ this guarantees that the sorted $\text{iEP}(P,i,N)$ has the desired structure. If we combine every two subsequent borders δ_l, δ_u in our sorted sequence $[\ldots,\delta_l,\delta_u,\ldots]$, then we receive our partition of intervals $\text{iPart}(P,i,N)$. For instance, if $x < 5$ and $x = 0$ are the only variable bounds in $\text{conIneqs}(P,i,N)$, then $\text{iEP}(P,i,N) = \{5),[5,0),[0,0],(0,(-\infty,\infty)\}$ and if we sort and combine them we get $\text{iPart}(P,i,N) = \{(-\infty,0),[0,0],(0,5),[5,\infty)\}$.

After constructing $\text{iPart}(P,i,N)$, we can finally construct the set of test points $\text{tps}(P,i,N)$ for argument position (P,i). If $|\text{avals}(P,i,N)| \in \mathbb{N}$, i.e., we determined that (P,i) is finite, then $\text{tps}(P,i,N) = \text{avals}(P,i,N)$. If the argument position (P,i) is over a first-order sort $\mathcal{F}_i$, i.e., $\text{sort}(P,i) = \mathcal{F}_i$, then we should always be able to determine that (P,i) is finite because $\mathbb{F}_i$ is finite. If the argument position (P,i) is over an arithmetic sort, i.e., $\text{sort}(P,i) = \mathcal{R}$ or $\text{sort}(P,i) = \mathcal{Z}$, and our approximation could not determine that (P,i) is finite, then the test-point set $\text{tps}(P,i,N)$ for (P,i) consists of at most two points per interval $I \in \text{iPart}(P,i,N)$: one integer value $a_I \in I \cap \mathbb{Z}$ if I contains integers (i.e. if $I \cap \mathbb{Z} \neq \emptyset$) and one non-integer value $b_I \in I \setminus \mathbb{Z}$ if I contains non-integers (i.e. if I is not just one integer point). Additionally, we enforce that $\text{tps}(P,i,N) = \text{tps}(Q,j,N)$ if $\text{conArgs}(P,i,N) = \text{conArgs}(Q,j,N)$ and both (P,i) and (Q,j) are infinite argument positions. (In our implementation of this test-point scheme, we optimize the test point selection even further by picking only one test point per interval—if possible an integer value and otherwise a non-integer—if all $\text{conArgs}(P,i,N)$ and all variables x connecting them in N have the same sort. However, we do not prove this optimization explicitly here because the proofs are almost identical to the case for two test points per interval.)

Based on these sets, we can now also define a tp-function β and an ep-function η. For the tp-function, we simply assign any argument position to $\text{tps}(P,i,N)$, i.e., $\beta(P,i) = \text{tps}(P,i,N) \cap \text{sort}(P,i)^{\mathcal{A}}$. (The intersection with $\text{sort}(P,i)^{\mathcal{A}}$ is needed to guarantee that the test-point set of an integer argument position is well-typed.) This also means that β is total and finite. For the ep-function η, we extrapolate any test-point vector $\bar{a}$ (with $\bar{a} = \bar{x}\sigma$ and $\sigma \in \text{wtis}_\beta(P(\bar{x}))$) over the (non-)integer subset of the intervals the test points belong to, i.e., $\eta(P,\bar{a}) = I'_1 \times \ldots \times I'_n$, where $I'_i = \{a_i\}$ if we determined that (P,i) is finite and otherwise I_i is the interval $I_i \in \text{iPart}(P,i,N)$ with $a_i \in I_i$ and $I'_i = I_i \cap \mathbb{Z}$ if a_i is an integer value and $I'_i = I_i \setminus \mathbb{Z}$ if a_i is a non-integer value. Note that this means that η might not be complete for every predicate P, e.g., when P has a finite argument position (P,i) with an infinite domain. However, both β and η together still cover the clause set N, cover any universal conjecture $N \models \forall\bar{x}.Q(\bar{x})$, and cover any existential conjecture $N \models \exists\bar{x}.Q(\bar{x})$.

Theorem 10. *The* tp-function *β covers N. The* tp-function *β covers an existential conjecture $N \models \exists\bar{x}.Q(\bar{x})$. The* tp-function *$\beta$ covers a universal conjecture $N \models \forall\bar{x}.Q(\bar{x})$.*

Example 11. Continuation of example 6: The majority of argument positions in our example are finite. Hence, determining their test point set is equivalent to the over-approximation of derivable values avals we computed for them: $\beta(\text{SpeedTable},1) = \{0,2000,4000,6000\}$, $\beta(\text{SpeedTable},2) = \{2000,4000,6000,8000\}$, $\beta(\text{SpeedTable},3) = \{1350,1600,1850,2100\}$, and $\beta(\text{IgnDeg},2) = \{1350,1600,1850,2100\}$. The other argument positions are all connected to $(\text{Speed},1)$ and $\text{conIneqs}(\text{Speed},1,N) = \{x_p < 0, x_p < 2000, x_p < 4000, x_p < 6000, x_p < 8000, x_p \geq 0, x_p \geq 2000, x_p \geq 4000, x_p \geq 6000, x_p \geq 8000\}$, from which we can compute

$$\text{iPart}(P,i,N) = \{(-\infty,0),[0,2000),[2000,4000),[4000,6000),[6000,8000),[8000,\infty)\}$$

and select the test point sets $\beta(\text{Speed},1) = \beta(\text{IgnDeg},1) = \beta(\text{ResArgs},1) = \beta(\text{Conj},1) = \{-1,0,2000,4000,6000,8000\}$. (Note that all variables in our problem are over the reals, so we only have to select one test point per interval! Moreover, in our previous version of the test point scheme, there would have been more intervals in the partition because we would have processed all inequalities, e.g., also those in $\text{conIneqs}(\text{SpeedTable},3,N)$.) The ep-function η that determines which interval is represented by which test point is $\eta(P,1,-1) = (-\infty,0)$, $\eta(P,1,0) = [0,2000)$, $\eta(P,1,2000) = [2000,4000)$, $\eta(P,1,4000) = [4000,6000)$, $\eta(P,1,6000) = [6000,8000)$, $\eta(P,1,8000) = [8000,\infty)$ for the predicates Speed, IgnDeg, ResArgs, and Conj. η behaves like the identity function for all other argument positions because they are finite.

From a Test-Point Function to a Datalog Hammer We can use the covering definitions, e.g., $\text{gnd}_\beta(N)$ is equisatisfiable to N, to instantiate our clause set (and conjectures) with numbers. As a result, we can simply evaluate all theory atoms and thus reduce our HBS(SLA)PA clause sets/conjectures to ground HBS clause sets, which means we could reduce our input into formulas without any arithmetic theory that can be solved by any Datalog reasoner. There is, however, one problem. The set $\text{gnd}_\beta(N)$ grows exponentially with regard to the maximum number of variables n_C in any clause in N, i.e. $O(|\text{gnd}_\beta(N)|) = O(|N| \cdot |B|^{n_C})$, where $B = \max_{(P,i)}(\beta(P,i))$ is the largest test-point set for any argument position. Since n_C is large for realistic examples, e.g., in our examples the size of n_C ranges from 9 to 11 variables, the finite abstraction is often too large to be solvable in reasonable time. Due to this blow-up, we have chosen an alternative approach for our Datalog hammer. This hammer exploits the ideas behind the covering definitions and will allow us to make the same ground deductions, but instead of grounding everything, we only need to (i) ground the negated conjecture over our tp-function and (ii) provide a set of ground facts that define which theory atoms are satisfied by our test points. As a result, the hammered formula is much more concise and we need no actual theory reasoning to solve the formula. In fact, we can solve the hammered formula by greedily applying unit resolution until this produces the empty clause—which would mean the conjecture is implied—or until it produces no more new facts—which would mean we have found a counter example. In practice, greedily applying resolution is not the best strategy and we recommend to use more advanced HBS techniques for instance those used by a state-of-the-art Datalog reasoner.

The Datalog hammer takes as input (i) an HBS(SLA)PA clause set N and (ii) optionally a universal conjecture $\forall \bar{y}.P(\bar{y})$. The case for existential conjectures is handled by encoding the conjecture $N \models \exists \bar{x}.Q(\bar{x})$ as the clause set $N \cup \{Q(\bar{x}) \rightarrow \bot\}$, which is unsatisfiable if and only if the conjecture holds. Given this input, the Datalog hammer first computes the tp-function β and the ep-function η as described above. Next, it computes four clause sets that will make up the Datalog formula. The first set $\text{tren}_N(N)$ is computed by abstracting away any arithmetic from the clauses $(\Lambda \parallel \Delta \rightarrow H) \in N$. This is done by replacing each theory atom A in Λ with a literal $P_A(\bar{x})$, where $\text{vars}(A) = \text{vars}(\bar{x})$ and P_A is a fresh predicate. The abstraction of the theory atoms is necessary because Datalog does not support non-constant function symbols (e.g., +,−) that would otherwise appear in approximately grounded theory atoms. Moreover, it is necessary to add extra sort literals $\neg Q_{(P,i,S)}(x)$ for some of the variables $x \in \text{vars}(H)$, where $H = P(\bar{t})$, $t_i = x$, $\text{sort}(x) = S$, and $Q_{(P,i,S)}$ is a fresh predicate. This is necessary in order to define the test point set for x if x does not appear in Λ or in Δ. It is also necessary in order to filter out any test points that are not integer values if x is an integer variable (i.e. $\text{sort}(x) = \mathcal{Z}$) but connected only to real sorted argument positions in Δ (i.e. $\text{sort}(Q,j) = \mathcal{R}$ for all $(Q,j) \in \text{depend}(x,\Delta)$). It is possible to reduce the number of fresh predicates needed, e.g., by reusing the same predicate for two theory atoms whose variables range over the same sets of test points. The resulting abstracted clause has then the form $\Delta_T, \Delta_S, \Delta \rightarrow H$, where Δ_T contains the abstracted theory literals (e.g. $P_A(\bar{x}) \in \Delta_T$) and Δ_S the "sort" literals (e.g. $Q_{(P,i,S)}(x) \in \Delta_S$). The second set is denoted by N_C and it is empty if we have no universal conjecture or if η does not cover our conjecture. Otherwise, N_C contains the ground and negated version ϕ of our universal conjecture $\forall \bar{y}.P(\bar{y})$. ϕ has the form $\Delta_\phi \rightarrow \bot$, where $\Delta_\phi = \text{gnd}_\beta(P(\bar{y}))$ contains all literals $P(\bar{y})$ for all groundings over β. We cannot skip this grounding but the worst-case size of Δ_ϕ is $O(\text{gnd}_\beta(P(\bar{y}))) = O(|B|^{n_\phi})$, where $n_\phi = |\bar{y}|$, which is in our applications typically much smaller than the maximum number of variables n_C contained in some clause in N. The third set is denoted by $\text{tfacts}(N,\beta)$ and contains a fact $\text{tren}_N(A)$ for every ground theory atom A contained in the theory part Λ of a clause $(\Lambda \,\|\, \Delta \rightarrow H) \in \text{gnd}_\beta(N)$ such that A simplifies to true. This is enough to ensure that our abstracted theory predicates evaluate every test point in every satisfiable interpretation $\mathcal{A}$ to true that also would have evaluated to true in the actual theory atom. Alternatively, it is also possible to use a set of axioms and a smaller set of facts and let the Datalog reasoner compute all relevant theory facts for itself. The set $\text{tfacts}(N,\beta)$ can be computed without computing $\text{gnd}_\beta(N)$ if we simply iterate over all theory atoms A in all constraints Λ of all clauses $Y = \Lambda \,\|\, \Delta \rightarrow H$ (with $Y \in N$) and compute all well typed groundings $\tau \in \text{wtis}_\beta(Y)$ such that $A\tau$ simplifies to true. This can be done in time $O(\mu(n_v) \cdot n_L \cdot |B|^{n_v})$ and the resulting set $\text{tfacts}(N,\beta)$ has worst-case size $O(n_A \cdot |B|^{n_v})$, where n_L is the number of literals in N, n_v is the maximum number of variables $|\text{vars}(A)|$ in any theory atom A in N, n_A is the number of different theory atoms in N, and $\mu(x)$ is the time needed to simplify a theory atom over x variables to a variable bound. The last set is denoted by $\text{sfacts}(N,\beta)$ and contains a fact $Q_{(P,i,S)}(a)$ for every fresh sort predicate $Q_{(P,i,S)}$ added during abstraction and every $a \in \beta(P,i) \cap S^{\mathcal{A}}$. This is enough to ensure that $Q_{(P,i,S)}$ evaluates to true for every test point assigned to the argument position (P,i) filtered by the sort S. Please note that already satifiability testing for BS clause sets is NEXPTIME-complete in general, and DEXPTIME-complete for the Horn case [26,33]. So when abstracting to a polynomially decidable clause set (ground HBS) an exponential factor is unavoidable.

Lemma 12. *N is equisatisfiable to its hammered version* $\text{tren}_N(N) \cup \text{tfacts}(N,\beta) \cup \text{sfacts}(N,\beta)$. *The conjecture $N \models \exists \bar{y}.Q(\bar{y})$ is false iff $N_D = \text{tren}'_N(N') \cup \text{tfacts}(N',\beta) \cup \text{sfacts}(N',\beta)$ is satisfiable with $N' = N \cup \{Q(\bar{y}) \rightarrow \bot\}$. The conjecture $N \models \forall \bar{y}.Q(\bar{y})$ is false iff $N_D = \text{tren}_N(N) \cup \text{tfacts}(N,\beta) \cup \text{sfacts}(N,\beta) \cup N_C$ is satisfiable.*

Note that $\text{tren}_N(N) \cup \text{tfacts}(N,\beta) \cup \text{sfacts}(N,\beta) \cup N_C$ is only a HBS clause set over a finite set of constants and not yet a Datalog input file. It is well known that such a formula can be transformed easily into a Datalog problem by adding a nullary predicate Goal and adding it as a positive literal to any clause without a positive literal. Querying for the Goal atom returns true if the HBS clause set was unsatisfiable and false otherwise.

Example 13. The hammered formula for example 6 looks as follows. The set of renamed clauses $\text{tren}_N(N)$ consists of all the previous clauses in N, except that inequalities have been abstracted to new first-order predicates:
$D'_1 : \text{SpeedTable}(0{,}2000{,}1350)$, $D'_2 : \text{SpeedTable}(2000{,}4000{,}1600)$,
$D'_3 : \text{SpeedTable}(4000{,}6000{,}1850)$, $D'_4 : \text{SpeedTable}(6000{,}8000{,}2100)$,
$C'_1 : P_{0 \le x_p}(x_p), P_{x_p < 8000}(x_p) \rightarrow \text{Speed}(x_p)$,
$C'_2 : P_{x_1 \le x_p}(x_1,x_p), P_{x_p < x_2}(x_p,x_2), \text{Speed}(x_p), \text{SpeedTable}(x_1,x_2,y) \rightarrow \text{IgnDeg}(x_p,y)$,
$C'_3 : \text{IgnDeg}(x_p,z) \rightarrow \text{ResArgs}(x_p)$, $C'_4 : \text{ResArgs}(x_p) \rightarrow \text{Conj}(x_p)$,
$C'_5 : P_{x_p \ge 8000}(x_p) \rightarrow \text{Conj}(x_p)$, $C'_6 : P_{x_p < 0}(x_p) \rightarrow \text{Conj}(x_p)$,
The set $\text{tfacts}(N,\beta)$ defines for which test points those new predicates evaluate to true:
$\{P_{0 \le x_p}(0)$, $P_{0 \le x_p}(2000)$, $P_{0 \le x_p}(4000)$, $P_{0 \le x_p}(6000)$, $P_{0 \le x_p}(8000)$, $P_{x_p < 8000}(-1)$,
$P_{x_p < 8000}(0)$, $P_{x_p < 8000}(2000)$, $P_{x_p < 8000}(4000)$, $P_{x_p < 8000}(6000)$, $P_{x_1 \le x_p}(0{,}0)$,
$P_{x_1 \le x_p}(0{,}2000)$, $P_{x_1 \le x_p}(0{,}4000)$, $P_{x_1 \le x_p}(0{,}6000)$, $P_{x_1 \le x_p}(0{,}8000)$, $P_{x_1 \le x_p}(2000{,}2000)$,
$P_{x_1 \le x_p}(2000{,}4000)$, $P_{x_1 \le x_p}(2000{,}6000)$, $P_{x_1 \le x_p}(2000{,}8000)$, $P_{x_1 \le x_p}(4000{,}4000)$,
$P_{x_1 \le x_p}(4000{,}6000)$, $P_{x_1 \le x_p}(4000{,}8000)$, $P_{x_1 \le x_p}(6000{,}6000)$, $P_{x_1 \le x_p}(6000{,}8000)$,
$P_{x_p < x_2}(-1{,}2000)$, $P_{x_p < x_2}(0{,}2000)$, $P_{x_p < x_2}(-1{,}4000)$, $P_{x_p < x_2}(0{,}4000)$,
$P_{x_p < x_2}(2000{,}4000)$, $P_{x_p < x_2}(-1{,}6000)$, $P_{x_p < x_2}(0{,}6000)$, $P_{x_p < x_2}(2000{,}6000)$,

Problem	Q	Status	$\lvert N\rvert$	vars	$\lvert B^m\rvert$	$\lvert\Delta_\phi\rvert$	SSPL	$\lvert B^s\rvert$	$\lvert\Delta^o_\phi\rvert$	SSPL06	vampire	spacer	z3	cvc4
lc_e1	∃	true	139	9	9	0	**< 0.1s**	45	0	**< 0.1s**	**< 0.1s**	**< 0.1s**	0,1	**< 0.1s**
lc_e2	∃	false	144	9	9	0	**< 0.1s**	41	0	**< 0.1s**	**< 0.1s**	**< 0.1s**	-	-
lc_e3	∃	false	138	9	9	0	**< 0.1s**	37	0	**< 0.1s**	**< 0.1s**	**< 0.1s**	-	-
lc_e4	∃	true	137	9	9	0	**< 0.1s**	49	0	**< 0.1s**	**< 0.1s**	**< 0.1s**	**< 0.1s**	**< 0.1s**
lc_e5	∃	false	152	13	9	0	33.5s	-	-	N/A	**< 0.1s**	-	-	-
lc_e6	∃	true	141	13	9	0	42.8s	-	-	N/A	**0.1s**	3.3s	11.5s	0.4s
lc_e7	∃	false	141	13	9	0	41.4s	-	-	N/A	**< 0.1s**	7.6s	-	-
lc_e8	∃	false	141	13	9	0	32.5s	-	-	N/A	**< 0.1s**	2.1s	-	-
lc_u1	∀	false	139	9	9	27	**< 0.1s**	45	27	**< 0.1s**	**< 0.1s**	N/A	-	-
lc_u2	∀	false	144	9	9	27	**< 0.1s**	41	27	**< 0.1s**	**< 0.1s**	N/A	-	-
lc_u3	∀	true	138	9	9	27	**< 0.1s**	37	27	**< 0.1s**	**< 0.1s**	N/A	**< 0.1s**	**< 0.1s**
lc_u4	∀	false	137	9	9	27	**< 0.1s**	49	27	**< 0.1s**	**< 0.1s**	N/A	-	-
lc_u5	∀	false	154	13	9	3888	32.4s	-	-	N/A	**0.1s**	N/A	-	-
lc_u6	∀	true	154	13	9	3888	32.5s	-	-	N/A	**2.3s**	N/A	-	-
lc_u7	∀	true	141	13	9	972	32.3s	-	-	N/A	**0.2s**	N/A	-	-
lc_u8	∀	false	141	13	9	1259712	**48.8s**	-	-	N/A	2351.4s	N/A	-	-
ecu_e1	∃	false	757	10	96	0	**< 0.1s**	624	0	1.3s	0.2s	0.1s	-	-
ecu_e2	∃	true	757	10	96	0	**< 0.1s**	624	0	1.3s	0.2s	0.1s	1.4s	0.4s
ecu_e3	∃	false	775	11	196	0	50.1s	660	0	41.5s	3.1s	**0.1s**	-	-
ecu_u1	∀	true	756	11	96	37	**0.1s**	620	306	1.1s	32.8s	N/A	197.5s	0.4s
ecu_u2	∀	false	756	11	96	38	**0.1s**	620	307	1.1s	32.8s	N/A	-	-
ecu_u3	∀	true	745	9	88	760	**< 0.1s**	576	11360	0.7s	1.2s	N/A	239.5s	0.1s
ecu_u4	∀	true	745	9	486	760	**< 0.1s**	2144	237096	15.9s	1.2s	N/A	196.0s	0.1s
ecu_u5	∀	true	767	10	96	3900	**0.1s**	628	415296	31.9s	-	N/A	-	-
ecu_u6	∀	false	755	10	95	3120	**< 0.1s**	616	363584	14.4s	597.8	N/A	-	-
ecu_u7	∀	false	774	11	196	8400	**48.9s**	656	2004708	-	-	N/A	-	-
ecu_u8	∀	true	774	11	196	8400	**48.7s**	656	2004708	-	-	N/A	-	-

Fig. 2. Benchmark results and statistics

$P_{x_p<x_2}(4000,6000)$, $P_{x_p<x_2}(-1,8000)$, $P_{x_p<x_2}(0,8000)$, $P_{x_p<x_2}(2000,8000)$,
$P_{x_p<x_2}(4000,8000)$, $P_{x_p<x_2}(6000,8000)$, $P_{x_p\geq 800}(8000)$, $P_{x_p<0}(-1)\}$
$\text{sfacts}(N,\beta)=\emptyset$ because there are no fresh sort predicates. The hammered negated conjecture is $N_C := \text{Conj}(-1), \text{Conj}(0), \text{Conj}(2000), \text{Conj}(4000), \text{Conj}(6000), \text{Conj}(8000) \rightarrow \perp$ and lets us derive false if and only if we can derive $\text{Conj}(a)$ for all test points $a \in \beta(\text{Conj},1)$.

4 Implementation and Experiments

We have implemented the sorted Datalog hammer as an extension to the SPASS-SPL system [11] (option -d) (SSPL in the table). By default the resulting formula is then solved with the Datalog reasoner VLog. The previously file-based combination with the Datalog reasoner VLog has been replaced by an integration of VLog into SPASS-SPL via the VLog API. We focus here only on the sorted extension and refer to [11] for an introduction into coupling of the two reasoners. Note that the sorted Datalog hammer itself is not fine tuned towards the capabilities of a specific Datalog reasoner nor VLog towards the sorted Datalog hammer.

In order to test the progress in efficiency of our sorted hammer, we ran the benchmarks of the lane change assistant and engine ECU from [11] plus more sophisticated, extended formalizations. While for the ECU benchmarks in [11] we modeled ignition timing computation adjusted by inlet temperature measurements, the new benchmarks take also gear box protection mechanisms into account. The lane change examples in [11] only simulated the supervisor for lane change assistants over some real-world instances. The new lane change benchmarks check properties for all potential inputs. The universal ones check that any suggested action by a lane change assistant is either proven as correct or disproven by our supervisor. The existential ones check safety properties, e.g., that the supervisor never returns both a proof and a disproof for the same input. We actually used SPASS-SPL to debug a prototype supervisor for lane change assistants during its development. The new lane change examples are based on versions generated during this debugging process where SPASS-SPL found the following bugs: (i) it did not always return a result, (ii) it declared actions as both safe and unsafe at the same time, and (iii) it declared actions as safe although they would lead to collisions. The supervisor is now fully verified.

The names of the problems are formatted so the lane change examples start with lc and the ECU examples start with ecu. Our benchmarks are prototypical for the complexity of HBS(SLA) reasoning in that they cover all abstract relationships between conjectures and HBS(SLA) clause sets. With respect to our two case studies we have many more examples showing respective characteristics. We would have liked to run benchmarks from other sources, but could not find any problems in the SMT-LIB [5,35] or CHC-COMP [2] benchmarks within the range of what our hammer can currently accept. Either the arithmetic part goes beyond SLA or there are further theories involved such as equality on first-order symbols.

For comparison, we also tested several state-of-the-art theorem provers for related logics (with the best settings we found): SPASS-SPL-v0.6 (SSPL06 in the table) that uses the original version of our Datalog Hammer [11] with settings `-d` for existential and `-d -n` for universal conjectures; the satisfiability modulo theories (SMT) solver *cvc4-1.8* [4] with settings `--multi-trigger-cache --full-saturate-quant`; the SMT solver *z3-4.8.12* [28] with its default settings; the constrained horn clause (CHC) solver *spacer* [24] with its default settings; and the first-order theorem prover *vampire-4.5.1* [37] with settings `--memory_limit 8000 -p off`, i.e., with memory extended to 8GB and without proof output. For the SMT/CHC solvers, we directly transformed the benchmarks into their respective formats. Vampire gets the same input as VLog transformed into the TPTP format [39]. Our experiments with vampire investigate how superposition reasoners perform on the hammered benchmarks compared to Datalog reasoners.

For the experiments, we used the TACAS 22 artifact evaluation VM (Ubuntu 20.04 with 8 GB RAM and a single processor core) on a system with an Intel Core i7-9700K CPU with eight 3.60GHz cores. Each tool got a time limit of 40 minutes for each problem.

The table in Fig. 2 lists for each benchmark problem: the name of the problem (Problem); the type of conjecture (Q), i.e., whether the conjecture is existential ∃ or universal ∀; the status of the conjecture (Status); number of clauses ($|N|$); maximum number of variables in a clause (vars); the size of the largest test-point set introduced by the sorted/original Hammer (B^s/B^o); the size of the hammered universal conjecture ($|\Delta_\phi|/|\Delta_\phi^o|$ for sorted/original); the remaining columns list the time needed by the tools to solve the benchmark problems. An entry "N/A" means that the benchmark example cannot be expressed in the tools input format, e.g., it is not possible to encode a universal conjecture (or, to be more precise, its negation) in the CHC format and SPASS-SPL-v0.6 is not sound when the problem contains integer variables. An entry "-" means that the tool ran out of time, ran out of memory, exited with an error or returned unknown.

The experiments show that SPASS-SPL (with the sorted Hammer) is orders of magnitudes faster than SPASS-SPL-v0.6 (with the original Hammer) on problems with universal conjectures. On problems with existential conjectures, we cannot observe any major performance gain compared to the original Hammer. Sometimes SPASS-SPL-v0.6 is even slightly faster (e.g. ecu_e3). Potential explanations are: First, the number of test points has a much larger impact on universal conjectures because the size of the hammered universal conjecture increases exponentially with the number of test points. Second, our sorted Hammer needs to generate more abstracted theory facts than the original Hammer because the latter can reuse abstraction predicates for theory atoms that are identical upto variable renaming. The sorted Hammer can reuse the same predicate only if variables also range over the same sets of test points, which we have not yet implemented.

Compared to the other tools, SPASS-SPL is the only one that solves all problems in reasonable time. It is also the only solver that can decide in reasonable time whether a universal conjecture is *not* a consequence. This is not surprising because to our knowledge SPASS-SPL is the only theorem prover that implements a decision procedure for HBS(SLA). On the problems with existential conjectures, our tool-chain solves all of the problems in under a minute and with comparable times to the best tool for the problem. The only exception are problems that contain a lot of superfluous clauses, i.e., clauses that are not needed to confirm/refute the conjecture. The reason might be that VLog derives all facts for the input problem in a breadth-first way, which is not very efficient if there are a lot of superfluous clauses. Vampire coupled with our sorted Hammer returns the best results for those problems. Vampire performed best on the hammered problems among all first-order theorem provers we tested, including iProver [25], E [38], and SPASS [40]. We tested all provers in default theorem proving mode with adjusted memory limits. The experiments with the first-order provers showed that our hammer also works reasonably well for them, but they do not scale well if the size and the complexity of the universal conjectures increases. For problems with existential conjectures, the CHC solver spacer is often the best, but as a trade-off it is unable to handle universal conjectures. The instantiation techniques employed by cvc4 are good for proving some universal conjectures, but both SMT solvers seem to be unable to disprove conjectures.

5 Conclusion

We have presented an extension of our previous Datalog hammer [11] supporting a more expressive input logic resulting in more elegant and more detailed supervisor formalizations, and through a soft typing discipline supporting more efficient reasoning. Our experiments show, compared to [11], that our performance on existential conjectures is at the same level as SMT and CHC solvers. The complexity of queries we can handle in reasonable time has significantly increased, see Section 4, Figure 2. Still SPASS-SPL is the only solver that can prove and disprove universal queries. The file interface between SPASS-SPL and VLog has been replaced by a close coupling resulting in a more comfortable application.

Our contribution here solves the third point for future work mentioned in [11] although there is still room to also improve our soft typing discipline. In the future, we want SPASS-SPL to produce explications that prove that its translations are correct. Another direction is to exploit specialized Datalog expressions and techniques, e.g., aggregation and stratified negation, to increase the efficiency of our tool-chain and to lift some restrictions from our input formulas. Finally, our hammer can be seen as part of an overall reasoning methodology for the class of BS(LA) formulas which we presented in [12]. We will implement and further develop this methodology and integrate our Datalog hammer.

Acknowledgments: This work was funded by DFG grant 389792660 as part of TRR 248 (CPEC), by BMBF in project ScaDS.AI, and by the Center for Advancing Electronics Dresden (cfaed). We thank our anonymous reviewers for their constructive comments.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1994)
2. Angelis, E.D., K, H.G.V.: Constrained horn clauses (chc) competition (2022), https://chc-comp.github.io/
3. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. Applicable Algebra in Engineering, Communication and Computing, AAECC **5**(3/4), 193–212 (1994)
4. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, LNCS, vol. 6806 (2011)
5. Barrett, C.W., de Moura, L.M., Ranise, S., Stump, A., Tinelli, C.: The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In: Barner, S., Harris, I.G., Kroening, D., Raz, O. (eds.) Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6504, p. 3. Springer (2010)
6. Baumgartner, P., Waldmann, U.: Hierarchic superposition revisited. In: Lutz, C., Sattler, U., Tinelli, C., Turhan, A., Wolter, F. (eds.) Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 11560, pp. 15–56. Springer (2019)
7. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 9300, pp. 24–51. Springer (2015)
8. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6173, pp. 107–121. Springer (2010)
9. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., González, L., Krötzsch, M., Marx, M., Murali, H.K., Weidenbach, C.: Artifact for a sorted Datalog hammer for supervisor verification conditions modulo simple linear arithmetic (Jan 2022). https://doi.org/10.5281/zenodo.5888272
10. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., González, L., Krötzsch, M., Marx, M., Murali, H.K., Weidenbach, C.: A sorted Datalog hammer for supervisor verification conditions modulo simple linear arithmetic. CoRR **abs/2201.09769** (2022), https://arxiv.org/abs/2201.09769
11. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., Krötzsch, M., Weidenbach, C.: A Datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In: Reger, G., Konev, B. (eds.) Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, United Kongdom, September 8-10, 2021. Proceedings. Lecture Notes in Computer Science, vol. 12941, pp. 3–24. Springer (2021)
12. Bromberger, M., Fiori, A., Weidenbach, C.: Deciding the bernays-schoenfinkel fragment over bounded difference constraints by simple clause learning over theories. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12597, pp. 511–533. Springer (2021)
13. Carral, D., Dragoste, I., González, L., Jacobs, C., Krötzsch, M., Urbani, J.: VLog: A rule engine for knowledge graphs. In: Ghidini et al., C. (ed.) Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II). LNCS, vol. 11779, pp. 19–35. Springer (2019)
14. Cimatti, A., Griggio, A., Redondi, G.: Universal invariant checking of parametric systems with quantifier-free SMT reasoning. In: Proc. CADE-28 (2021), to appear
15. Downey, P.J.: Undecidability of presburger arithmetic with a single monadic predicate letter. Tech. rep., Center for Research in Computer Technology, Harvard University (1972)

16. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. **22**(3), 364–418 (1997)
17. Faqeh, R., Fetzer, C., Hermanns, H., Hoffmann, J., Klauck, M., Köhl, M.A., Steinmetz, M., Weidenbach, C.: Towards dynamic dependable systems through evidence-based continuous certification. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12477, pp. 416–439. Springer (2020)
18. Fiori, A., Weidenbach, C.: SCL with theory constraints. CoRR **abs/2003.04627** (2020), https://arxiv.org/abs/2003.04627
19. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 306–320. Springer (2009)
20. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 405–416. ACM (2012)
21. Hillenbrand, T., Weidenbach, C.: Superposition for bounded domains. In: Bonacina, M.P., Stickel, M. (eds.) McCune Festschrift. LNCS, vol. 7788, pp. 68–100. Springer (2013)
22. Horbach, M., Voigt, M., Weidenbach, C.: On the combination of the bernays-schönfinkel-ramsey fragment with simple linear integer arithmetic. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 77–94. Springer (2017)
23. Horbach, M., Voigt, M., Weidenbach, C.: The universal fragment of presburger arithmetic with unary uninterpreted predicates is undecidable. CoRR **abs/1703.01212** (2017)
24. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 17–34. Springer (2014)
25. Korovin, K.: iprover - an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5195, pp. 292–298. Springer (2008)
26. Lewis, H.R.: Complexity results for classes of quantificational formulas. Journal of Compututer and System Sciences **21**(3), 317–353 (1980)
27. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal **36**(5), 450–462 (1993)
28. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 4963 (2008)
29. de Moura, L.M., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Communications of the ACM **54**(9), 69–77 (2011)
30. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). Journal of the ACM **53**, 937–977 (November 2006)
31. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
32. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Handbook of Automated Reasoning, pp. 335–367. Elsevier and MIT Press (2001)
33. Plaisted, D.A.: Complete problems in the first-order predicate calculus. Journal of Computer and System Sciences **29**, 8–35 (1984)
34. Ranise, S.: On the verification of security-aware e-services. Journal of Symbolic Compututation **47**(9), 1066–1088 (2012)

35. Ranise, S., Tinelli, C., Barrett, C., Fontaine, P., Stump, A.: Smt-lib the satisfiability modulo theories library (2022), https://smtlib.cs.uiowa.edu/
36. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 112–131. Springer (2018)
37. Riazanov, A., Voronkov, A.: The design and implementation of vampire. AI Communications **15**(2-3), 91–110 (2002)
38. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) Proc. of the 27th CADE, Natal, Brasil. pp. 495–507. No. 11716 in LNAI, Springer (2019)
39. Sutcliffe, G.: The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. J. Autom. Reason. **59**(4), 483–502 (2017)
40. Weidenbach, C., Dimova, D., Fietzke, A., Suda, M., Wischnewski, P.: Spass version 3.5. In: Schmidt, R.A. (ed.) 22nd International Conference on Automated Deduction (CADE-22). Lecture Notes in Artificial Intelligence, vol. 5663, pp. 140–145. Springer, Montreal, Canada (August 2009)