Relaxed Stratification: A New Approach to Practical Complete Predicate Refinement[*]

Tachio Terauchi[1] and Hiroshi Unno[2]

[1] JAIST
terauchi@jaist.ac.jp
[2] University of Tsukuba
uhiro@cs.tsukuba.ac.jp

Abstract. In counterexample-guided abstraction refinement, a predicate refinement scheme is said to be *complete* for a given theory if it is guaranteed to eventually find predicates sufficient to prove the given property, when such exist. However, existing complete methods require deciding if a proof of the counterexample's spuriousness exists in some finite language of predicates. Such an *exact* finite-language-restricted predicate search is quite hard for many theories used in practice and incurs a heavy overhead. In this paper, we address the issue by showing that the language restriction can be relaxed so that the refinement process is restricted to infer proofs from some finite language $L_{base} \cup L_{ext}$ but is only required to return a proof when the counterexample's spuriousness can be proved in L_{base}. Then, we show how a proof-based refinement algorithm can be made to satisfy the relaxed requirement and be complete by restricting only the theory-level reasoning in SMT to emit L_{base}-restricted partial interpolants (while such an approach has been proposed previously, we show for the first time that it can be done for languages that are not closed under conjunctions and disjunctions). We also present a technique that uses a property of counterexample patterns to further improve the efficiency of the refinement algorithm while still satisfying the requirement. We have experimented with a prototype implementation of the new refinement algorithm, and show that it is able to achieve complete refinement with only a small overhead.

1 Introduction

Predicate abstraction with counterexample-guided abstraction refinement (CEGAR) is a promising approach to automated verification of safety (i.e., reachability) properties (see, e.g., [6] for a survey). Briefly, the CEGAR approach works as follows. Let $\mathcal{T}$ be a first-order logic (FOL) theory. The verifier picks some finite set of predicates from $\mathcal{T}$ as the initial *candidate predicate set*, and iterates the following two processes until convergence (here, we use the term "predicate" for an arbitrary formula, and not limited to just atomic predicates).

---

(1) The *abstraction process* checks if the current candidates form a sufficient proof of the program's safety (i.e., an inductive invariant – sometimes called "safe" inductive invariant). If so, then the program is proved safe and the iteration halts. Otherwise, the process generates a *counterexample* as an evidence that the current candidates are insufficient, and (2) is invoked.

(2) The *refinement process* analyzes the given counterexample. If the counterexample cannot be proved spurious by predicates from $\mathcal{T}$ (i.e., "the counterexample is real"), then the iteration halts and the program is detected to be unsafe. Otherwise, the predicates inferred as a proof of the counterexample's spuriousness are added to the candidates, and we repeat from (1).

Note that the verifier halts either when sufficient predicates are inferred to prove the program safe, or a real counterexample is discovered.

For an unsafe program, the state-of-the-art CEGAR-based verifiers are usually able to eventually discover a real counterexample and converge, by exploring the state space in a fair manner (if somewhat slowly for ones requiring large counterexamples). By contrast, when a program is safe and the underlying theory $\mathcal{T}$ is sufficient for proving the safety, most verifiers have no guarantee of convergence and can diverge by having the refinement process indefinitely produce incorrect candidate proofs.

For example, consider the C-like program shown in Figure 1. Here, `ndet()` returns a non-deterministic integer. The goal is to verify that the assertion failure is unreachable, that is, $a = b \Rightarrow y = x$ whenever line 10 is reached.

Suppose we start the verification process with the candidate set comprising the boolean closure of the predicates $z = 0$, $a = b$ and $y = x$. A possible counterexample is a path that passes through the first loop (lines 4-6) once, reaching line 7 with the abstract state $(a = b \Rightarrow y \neq x) \wedge z \neq 0$, and then passes through the second loop (lines 7-9) once, reaching line 10 with the abstract state $z = 0$, which does not

```
1:  void main(void) {
2:    int a = ndet();int b = ndet();
3:    int x = a;int y = b;int z = 0;
4:    while (ndet()) {
5:      y++;z++;
6:    }
7:    while (z != 0) {
8:      y--;z--;
9:    }
10:   if (a=b && y!=x) { assert false; }
11: }
```

Fig. 1. A program on which CEGAR may diverge.

imply $a = b \Rightarrow y = x$. A possible proof of the counterexample's spuriousness (i.e., proof that the path is actually safe) is the predicate $a = b \Rightarrow y = x + z$. The predicate turns out to be an inductive invariant for the program, and the verification process halts in the next iteration.

Unfortunately, the refinement process is not guaranteed to infer such a predicate but may choose any predicates that can prove the counterexample's safety. For instance, another possibility is the predicate $a = b \Rightarrow y = x + 1$. Adding this to the candidate set is sufficient for proving the safety of the counterexample

but not that of the program, and the abstraction process in the subsequent iteration would return yet another counterexample. For example, it may return the counterexample that passes the first loop twice, reaching line 7 with the abstract state $a = b \Rightarrow y \neq x + 1$, and then passes through the second loop twice, reaching line 10 with the abstract state $z = 0$ again. Then, the refinement process may choose the predicate $a = b \Rightarrow y = x + 2$ to prove the spuriousness of this new counterexample, which is still insufficient to prove the whole program correct. The abstract-and-refine iteration may repeat indefinitely in this manner, adding to the candidates the predicate $a = b \Rightarrow y = x + i$ in each i-th run of the refinement process. A refinement process is said to be *complete* (w.r.t. $\mathcal{T}$) if the CEGAR process is guaranteed to converge and eventually discover a proof of the program's safety, when one exists in $\mathcal{T}$.

Previous works [7, 11] have proposed to achieve complete refinement in CEGAR by stratifying $\mathcal{T}$ into an infinite sequence of predicate languages $L^0 \subseteq L^1 \subseteq \ldots L^k \ldots$ such that $\mathcal{T} = \bigcup_{k \in \omega} L^k$, and requiring each i-th run of the refinement to only infer predicates from the stratum $L^{lvl(i)}$ where $lvl(i)$ is the stratum level at the i-th CEGAR iteration. By requiring each L^k to be finite[3] and raising the stratum level just when the refinement process reports that no proof exists for the given counterexample in the current stratum, the approach guarantees completeness. However, the approach requires the refinement process to exactly decide if there is a proof of the given counterexample in the current stratum. Indeed, completeness would be lost if the refinement process was allowed to report that the current stratum does not have a proof when it actually does. For many theories used in practice, such as the theory of linear real arithmetic, such an exact finite-language-restricted proof search incurs heavy overhead and is prohibitive (see Section 4, Section 5, and the extended report [18] for analysis and discussion).

The first contribution of this paper is the observation that exact finite language restricted proof search is actually unnecessary for completeness. Instead, we show that the following more relaxed scheme is sufficient: in each i-th run of the refinement process, we restrict the returned proof to some finite language of predicates $L_{base}^{lvl(i)} \cup L_{ext}^{lvl(i)}$ (*base* and *extension*) such that the refinement process may report that no proof exists only when no proof exists in $L_{base}^{lvl(i)}$. There are no further restrictions on the refinement process, and so, the refinement process may return a proof that is not in L_{base} (but in $L_{base} \cup L_{ext}$) even if a proof exists in L_{base}, or may report that no proof exists even if a proof exists in $L_{base} \cup L_{ext}$ (but not in L_{base}). We show that this relaxed approach still ensures completeness when the stratum level is raised just when the refinement process reports that there is no proof in the current stratum, as before, and L_{base} grows to eventually cover $\mathcal{T}$ (i.e., $L_{base}^0 \subseteq L_{base}^1 \subseteq \ldots L_{base}^k \ldots$ such that $\mathcal{T} = \bigcup_{k \in \omega} L_{base}^k$). We formalize this observation in a refinement algorithm scheme called *relaxed stratification* (contra the *exact stratification* approach described above) and prove that it is indeed complete.

---

[3] The term "finite predicate language" is used synonymously with "finite set of predicates".

As the second contribution, we present a concrete refinement algorithm that implements the relaxed scheme. The algorithm is a modification of the proof-based refinement [5] in which the theory-level reasoning is restricted so that partial (tree-)interpolants at that level is restricted to L_{base}.[4] We also present a technique that uses a certain property of the counterexample patterns to further improve the efficiency of the algorithm while still satisfying the requirement of the scheme. We formalize the refinement algorithm as a constraint solver for recursion-free Horn-clause constraints [3, 12, 13, 21] which has gained popularity as the standard format for describing refinement algorithms. We have implemented a prototype of the refinement algorithm, and we show empirically that it is able to achieve complete predicate refinement with low overhead.

In summary, the paper's contributions are as follows:

- A new scheme for practical complete predicate refinement called *relaxed stratification* and the proof of its completeness (Section 2).
- A new predicate refinement algorithm as concrete instance of the relaxed stratification scheme (Section 3).
- Experiments with a prototype implementation of the refinement algorithm (Section 4).

The rest of the paper is organized as follows. Section 2 formally defines the relaxed stratification scheme and proves its completeness. Section 3 presents the concrete refinement algorithm implementing the scheme. Section 4 presents experimental results with the prototype implementation of the refinement algorithm. We discuss related work in Section 5 and conclude the paper in Section 6. Supplementary material contains the extended report with proofs and extra materials omitted from the main body of the paper, and the benchmarks used in the experiments [18].

2   The Relaxed Stratification Scheme

Let $\mathcal{T}$ be a FOL theory. For a formula θ in the signature of $\mathcal{T}$ (a $\mathcal{T}$-formula), we write $fvs(\theta)$ for the free variables in θ. A *predicate* in $\mathcal{T}$ is of the form $\lambda x_1, \ldots, x_n.\theta$ where θ is a $\mathcal{T}$-formula such that $fvs(\theta) \subseteq \{x_1, \ldots, x_n\}$. For readability, we often omit the explicit λ abstraction and treat a formula θ as the predicate $\lambda \bar{x}.\theta$ where $\{\bar{x}\} = fvs(\theta)$. We overload $\mathcal{T}$ for the set of predicates in $\mathcal{T}$.

2.1   Assumptions on the Abstraction Process

Relaxed stratification only concerns the refinement process part of CEGAR. We show that the scheme is quite general and can be used in a wide range of CEGAR-based verifiers. To this end, we delineate the conditions that the

---

[4] While this approach has already been suggested in [7], they require the restricting language to be closed under conjunctions and disjunctions (see Section 5 for further discussion).

abstraction process part needs to satisfy. As we shall show below, the conditions are quite weak and satisfied by virtually any CEGAR-based verifier.

We assume that the abstraction process Abs takes as input a program and a finite set of predicates in $\mathcal{T}$ (the set of *candidate proofs*). For a program M and a finite set of predicates $F \subseteq \mathcal{T}$, we require that $\mathsf{Abs}(M, F)$ either returns safe, indicating that M has been proved safe using the predicates from F, or returns a *counterexample*. For generality, we assume that a counterexample is also simply a program so that, for a counterexample M, we write $\mathsf{Abs}(M, F) = \mathsf{safe}$ when F is sufficient for the abstraction process to prove the spuriousness of M (in practice, a counterexample is not an arbitrary program, but, e.g., an unwound program slice of the input program, and concrete instances of the relaxed stratification scheme take advantage of the counterexample structure – cf. Section 3). We sometimes say that F *refutes* the counterexample M when $\mathsf{Abs}(M, F) = \mathsf{safe}$.

We require Abs to be monotonic on the candidates, that is, if $\mathsf{Abs}(M, F) = \mathsf{safe}$ and $F \subseteq F'$ then $\mathsf{Abs}(M, F') = \mathsf{safe}$ (i.e., having more predicates can only increase Abs's ability to prove). We also require that if $\mathsf{Abs}(M, F) = \mathsf{cex}(M')$ then $\mathsf{Abs}(M', F) \neq \mathsf{safe}$, that is, the returned counterexample is actually a counterexample and cannot be refuted by the given predicates. Finally, we require that if $\mathsf{Abs}(M, F) = \mathsf{safe}$ and $\mathsf{Abs}(M, F') = \mathsf{cex}(M')$ then $\mathsf{Abs}(M', F) = \mathsf{safe}$, that is, if a set of predicates is a proof for program's safety then it is also a proof for any counterexample of the program. We say that Abs is *sound* when it only proves safe programs safe, that is, $\mathsf{Abs}(M, F) = \mathsf{safe}$ only if M is safe.[5]

We note that the assumptions on the abstraction process are quite liberal and do not demand, for example, the process uses the given set of predicates by decomposing them into atomic predicates and taking their boolean closure, taking the cartesian closure, or using them directly as loop invariants. In Example 1 below, we describe an example abstraction process that uses the predicates directly.

Example 1. Let $\mathcal{T}$ be the quantifier-free theory of linear real arithmetic. Let Abs be the abstraction process that, given a program (or counterexample) M and the set of predicates $F \subseteq \mathcal{T}$, checks if there exists an assignment from each loop-head location in M to a predicate in F that forms an inductive invariant of M. Recall the example from Section 1. Let M_{ex} be the program shown in Figure 1. Then, the map ρ such that $\rho(\mathsf{L4}) = \rho(\mathsf{L7}) = \theta_{ex1}$ where $\theta_{ex1} \equiv a = b \Rightarrow y = x + z$ is an inductive invariant of M_{ex}, and therefore, $\mathsf{Abs}(M_{ex}, \{\theta_{ex1}\}) = \mathsf{safe}$.

When given an insufficient set of predicates as the candidates, Abs returns a counterexample. For instance, as discussed in Section 1, a possible counterexample of M_{ex} is M_{exa}, shown in Figure 2, that passes through each loop once to reach line 10. (The semantics of $\mathsf{assume}\ (b)$ is to safely halt if b is false, and proceed otherwise.) Here, $\mathsf{a1}$–$\mathsf{a4}$ label the entry points of the unwound loops. Viewing them as one-iteration loops where invariants are asserted, it can be seen that ρ such that $\rho(\ell) = \theta_{ex1}$ for each $\ell \in \{\mathsf{a1}$–$\mathsf{a4}\}$ is an inductive invariant of M_{exa}, and therefore $\mathsf{Abs}(M_{exa}, \{\theta_{ex1}\}) = \mathsf{safe}$. However, as discussed in

---

[5] Soundness of Abs is not required for completeness of relaxed stratification.

```
    int a=ndet();int b=ndet();              int a=ndet();int b=ndet();
    int x=a;int y=b;int z=0;                int x=a;int y=b;int z=0;
 a1:assume (ndet());                     b1:assume (ndet());
    y++;z++;                                y++;z++;
 a2:assume (ndet());                     b2:assume (ndet());
 a3:assume (z != 0);                        y++;z++;
    y--;z--;                             b3:assume (ndet());
 a4:assume (z == 0);                     b4:assume (z != 0);
    assume (a=b && y!=x);                   y--;z--;
    assert false;                        b5:assume (z != 0);
                                            y--;z--;
                                         b6:assume (z == 0);
                                            assume (a=b && y!=x);
                                            assert false;


            M_exa                                   M_exb
```

Fig. 2. Counterexamples of the program from Figure 1.

Section 1, asserting θ_0 at a1, a4, and θ_1 at a2, a3 where $\theta_0 \equiv a = b \Rightarrow y = x$ and $\theta_1 \equiv a = b \Rightarrow y = x + 1$ also constitutes a sufficient loop invariant of M_{exa}. Therefore, we also have $\mathsf{Abs}(M_{exa}, \{\theta_0, \theta_1\}) = \mathsf{safe}$.

Similarly, M_{exb} shown in Figure 2 is a counterexample that passes through each loop twice to reach line 10. By reasoning similar to the above, we have $\mathsf{Abs}(M_{exb}, \{\theta_{ex1}\}) = \mathsf{Abs}(M_{exb}, \{\theta_0, \theta_1, \theta_2\}) = \mathsf{safe}$ where $\theta_2 \equiv a = b \Rightarrow y = x + 2$. ▲

2.2 The Relaxed Stratification Scheme

We are now ready to formalize the relaxed stratification scheme. The core of the scheme is the relaxed finite-language-restricted refinement process RlxRef that takes as input a counterexample and a *restricting predicate language* (L_{base}, L_{ext}), and returns either unsafe indicating that the counterexample is real, a set of predicates $F \subseteq L_{base} \cup L_{ext}$ that proves the safety of the counterexample, or $\mathsf{noproof}$ indicating that it could not find a proof for the counterexample within the given restriction.

We prepare strata of restricting predicate languages:

$$(L_{base}^0, L_{ext}^0), \ (L_{base}^1, L_{ext}^1), \ \dots \ (L_{base}^k, L_{ext}^k), \ \dots$$

We require each restricting predicate language to be finite, and the base-part to eventually cover $\mathcal{T}$. Formally, we impose the following condition on the restricting predicate languages: 1.) for each $k \in \omega$, $L_{base}^k \cup L_{ext}^k$ is a finite subset of $\mathcal{T}$, 2.) for each $k \in \omega$, $L_{base}^k \subseteq L_{base}^{k+1}$, and 3.) $\mathcal{T} = \bigcup_{k \in \omega} L_{base}^k$.

Figure 3 shows the overview of the relaxed stratification verification process. The verification procedure $\mathsf{RlxCegar}$ takes as input the program M to be verified, and first initializes the candidate predicate set *Cands* to $\varnothing$ (line 2) and the restricting language stratum k to 0 (line 3). Then, it repeats the abstract-and-refine loop (lines 4-10) until convergence. The loop first calls $\mathsf{Abs}(M, Cands)$ to

```
01: RlxCegar(M) =
02:    Cands := ∅;
03:    k := 0;
04:    while true do
05:      match Abs(M,Cands) with
06:        safe → return safe
07:        | cex(M′) → match RlxRef(M′,L^k_{base},L^k_{ext}) with
08:                      unsafe → return unsafe
09:                      | prf(F) → Cands := Cands ∪ F
10:                      | noproof → k := k + 1
```

Fig. 3. The relaxed stratification verification process.

check if M can be proved safe with the current candidates. If so, then we exit the verification process, returning safe (line 6). Otherwise, a counterexample M' is obtained, and we call RlxRef on M' and the current restricting language (L^k_{base}, L^k_{ext}) (line 7). If RlxRef returns unsafe, then the counterexample is real and we exit the verification process, returning unsafe (line 8). Otherwise, RlxRef either returns a set of predicates that refutes the counterexample (line 9), or returns noproof indicating that it has failed to find a proof for the counterexample in the current language stratum (line 10). In the former case, the returned set of predicates are added to *Cands*, and in the latter case, the language stratum is raised to the next level.

We require RlxRef to only report unsafe on a real counterexample, that is, RlxRef(M, L_{base}, L_{ext}) = unsafe only if $\forall F \subseteq \mathcal{T}.\text{Abs}(M, F) \neq$ safe, and we require the returned proof to be actually a proof of the counterexample, that is RlxRef(M, L_{base}, L_{ext}) = prf(F) only if Abs(M, F) = safe (these conditions are not particular to relaxed stratification and usually assumed for any refinement process in CEGAR). In addition, we require RlxRef to only infer proofs from the given restricting predicate language and be able to return some proof if the given counterexample is refutable just in the base part of the language. Formally, we impose the following additional conditions on RlxRef:

– If RlxRef(M, L_{base}, L_{ext}) = prf(F) then $F \subseteq L_{base} \cup L_{ext}$; and
– If $\exists F \subseteq L_{base}.\text{Abs}(M, F)$ = safe, then RlxRef(M, L_{base}, L_{ext}) = prf(F') for some F'.

We state and prove the completeness of the relaxed stratification scheme.

Theorem 1 (Completeness). *If $\exists F \subseteq \mathcal{T}.\text{Abs}(M, F)$ = safe, then RlxCegar(M) terminates and returns safe.*

We remind that safety verification is undecidable in general, and our "completeness" only states that the verification terminates under the promise that a proof of the program's safety exists in $\mathcal{T}$.[6]

---

[6] This notion of completeness is the same as the one from previous works [7, 11].

Also, assuming that Abs is sound (cf. Section 2.1), it is easy to see that RlxCegar is also sound in that it only proves safe programs safe (in fact, this holds independently of the behavior of RlxRef).

Theorem 2 (Soundness). *If Abs is sound, then RlxCegar(M) returns* safe *only if M is safe.*

Example 2. We show how the relaxed stratification scheme would ensure the convergence of a verifier on the program M_{ex} from Example 1. Suppose that we run RlxCegar(M_{ex}), and for contradiction, it diverges by generating the following infinite series of refinements discussed in Section 1:

$$a = b \Rightarrow y = x, \quad a = b \Rightarrow y = x + 1, \quad \ldots \quad a = b \Rightarrow y = x + i, \quad \ldots$$

By the definition of RlxCegar, it must be the case that the restricting predicate language at the i-th CEGAR iteration is $(L_{base}^{lvl(i)}, L_{ext}^{lvl(i)})$ such that $y = x + i \in L_{base}^{lvl(i)} \cup L_{ext}^{lvl(i)}$ where $lvl(i)$ is the restricting language stratum level in the i-th iteration. Because each $L_{base}^{lvl(i)} \cup L_{ext}^{lvl(i)}$ is finite, the stratum level of the language must have been raised infinitely many times. Therefore, $a = b \Rightarrow y = x + z \in L_{base}^{j}$ for some j because $\mathcal{T} = \bigcup_{k \in \omega} L_{base}^{k}$.

But, as argued in Example 1, $a = b \Rightarrow y = x + z$ is a sufficient proof of M_{ex}'s safety, and therefore also that of its counterexamples. Therefore, by the fact that RlxRef refutes any counterexample refutable in the base part of the given restricting language, for any counterexample M' of M_{ex}, RlxRef($M', L_{base}^{j}, L_{ext}^{j}$) = prf($F'$) for some $F' \subseteq L_{base}^{j} \cup L_{ext}^{j}$. Then, because $L_{base}^{j} \cup L_{ext}^{j}$ is finite, RlxCegar must have eventually inferred a sufficient set of predicates that constitutes a proof of M_{ex}'s safety without further raising the language stratum. ▲

3 Concrete Refinement Algorithm Instances

We show how to implement the relaxed finite-language-restricted refinement process RlxRef. In fact, we describe a technique that takes as module an exact L_{base}-restricted refinement algorithm and turn it into a relaxed ($L_{base}, L_{base}^{\wedge\vee}$)-restricted refinement algorithm. (We write $L^{\wedge\vee}$ for the closure of L under conjunctions and disjunctions.) We focus on the case where the given counterexample is spurious.[7]

Following the recent trend [17, 3, 8, 2, 1, 13, 12, 21], we formalize the refinement algorithm as a constraint solver for recursion-free Horn-clause constraints. Specifically, we present a relaxed (L_{base}, L_{ext})-restricted constraint solver that takes as module an exact L_{base}-restricted constraint solver (cf. Section 3.1 for the definition of exact/relaxed finite-language-restricted constraint solvers). We review Horn-clause constraints in Section 3.1, and describe the constraint solver, that we call RlxSolveA, in Section 3.2.

---

[7] Detecting if the counterexample real and returning unsafe if so can be handled via usual unrestricted refinement (cf. Section 4).

We also present a technique that takes as module a relaxed (L_{base}, L_{ext})-restricted constraint solver, an unrestricted constraint solver $\mathcal{AU}$, and a positive integer parameter ℓ, and turn them into a relaxed $(L_{base}, \mathsf{LB}(L_{base} \cup L_{ext}, \mathcal{AU}, \ell))$-restricted constraint solver where $\mathsf{LB}(L, \mathcal{AU}, \ell)$ is a certain finite language of predicates determined by L, $\mathcal{AU}$, and ℓ. We formalize the technique as the constraint solver RlxSolveB, described in Section 3.3. The technique applies the relaxed finite-language-restricted constraint solver provided as the module to only a small subset of the constraint solving problem, and can be used to improve the efficiency of the given relaxed finite-language-restricted constraint solver.

We remind that the exact finite-language-restricted proof search is an inherently expensive process (cf. Section 5, Section 4, and the extended report [18]), and the key idea in these constraint solvers is to use the expensive exact finite-language-restricted proof search process (given as a module) only on small subparts of the problem. This is made possible thanks to the relaxed requirement on the language restriction where the refinement process is not required to exactly decide the existence of a restricted solution for the whole problem. Informally, the trick is to choose the subproblems just large enough to guarantee that if a subproblem is not L_{base} solvable then neither is the whole and that there can only be finitely many solutions for the whole obtainable from L_{base}-restricted solutions for the subproblems.

3.1 Horn Clause Constraints

For concreteness, in what follows, we assume that the underlying theory $\mathcal{T}$ is the quantifier-free theory of linear real arithmetic (QFLRA). However, the techniques presented in Sections 3.2 and 3.3 can be applied to any quantifier-free theory.

A *formula* θ in the signature of QFLRA comprises *atomic predicate p* of the form $a_1 x_1 + a_2 x_2 \cdots + a_n x_n \leq a_{n+1}$ where $a_1, \ldots, a_{n+1} \in \mathbb{Z}$, and is closed under the usual boolean operations $\neg$, $\wedge$, $\vee$, and $\Rightarrow$. As usual, we let $\neg$ bind the tightest and $\Rightarrow$ the weakest. A *literal l* is either an atomic predicate or its negation. A *clause* C is a disjunction of literals. A conjunctive normal form (CNF) is a conjunction of clauses. We often use a set to represent a clause or a CNF so that $\{l_1, \ldots, l_n\}$ represents $l_1 \vee \cdots \vee l_n$ and $\{C_1, \ldots, C_n\}$ represents $C_1 \wedge \cdots \wedge C_n$. We write $\bot$ for contradiction and $\top$ for tautology. We write $\vDash \theta$ when θ is valid in $\mathcal{T}$.

Horn Clauses and Horn-Clause Constraints. A *predicate variable application* is of the form $P(\bar{x})$ where P is a *predicate variable* of arity $|\bar{x}|$. A *Horn clause hc* is of the form $\theta \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$ where θ is a formula in $\mathcal{T}$, each B_i is a predicate variable application, and H is a predicate variable application or $\bot$. We call H the *head* of the Horn clause, and $\theta \wedge B_1 \wedge \cdots \wedge B_n$ the *body*. A Horn clause whose head is $\bot$ is called a *root clause*.

A *Horn-clause constraint set* (HCCS) $\mathcal{H}$ is a finite set of Horn clauses. We write $pvs(\mathcal{H})$ for the predicate variables in $\mathcal{H}$. We write $leaves(\mathcal{H})$ for the set of predicate variables in $\mathcal{H}$ that do not occur as a head in $\mathcal{H}$. We define $\rightsquigarrow_{\mathcal{H}}$ to be the relation $\{(P, Q) \mid \theta \wedge \ldots P(\bar{x}) \ldots \rightarrow Q(\bar{y}) \in \mathcal{H}\}$. We say that a Horn clause

$$
\begin{array}{lll}
\theta_{\mathrm{p1}} \dashrightarrow P(\bar{x}) & \theta_{\mathrm{p1}} \dashrightarrow P_1(\bar{x}) & \theta_{\mathrm{p1}} \dashrightarrow P_1(\bar{x}) \\
\theta_{\mathrm{p2}} \wedge P(\bar{x}) \dashrightarrow P(\bar{x}') & \theta_{\mathrm{p2}} \wedge P_1(\bar{x}) \dashrightarrow P_2(\bar{x}') & \theta_{\mathrm{p2}} \wedge P_1(\bar{x}) \dashrightarrow P_2(\bar{x}') \\
P(\bar{x}) \dashrightarrow Q(\bar{x}) & P_2(\bar{x}) \dashrightarrow Q_1(\bar{x}) & \theta_{\mathrm{p2}} \wedge P_2(\bar{x}) \dashrightarrow P_3(\bar{x}') \\
\theta_{\mathrm{p3}} \wedge Q(\bar{x}) \dashrightarrow Q(\bar{x}') & \theta_{\mathrm{p3}} \wedge Q_1(\bar{x}) \dashrightarrow Q_2(\bar{x}') & P_3(\bar{x}) \dashrightarrow Q_1(\bar{x}) \\
\theta_{\mathrm{p4}} \wedge Q(\bar{x}) \dashrightarrow \bot & \theta_{\mathrm{p4}} \wedge Q_2(\bar{x}) \dashrightarrow \bot & \theta_{\mathrm{p3}} \wedge Q_1(\bar{x}) \dashrightarrow Q_2(\bar{x}') \\
& & \theta_{\mathrm{p3}} \wedge Q_2(\bar{x}) \dashrightarrow Q_3(\bar{x}') \\
& & \theta_{\mathrm{p4}} \wedge Q_3(\bar{x}) \dashrightarrow \bot \\
\\
\qquad \mathcal{H}_{ex} & \qquad \mathcal{H}_{exa} & \qquad \mathcal{H}_{exb}
\end{array}
$$

Fig. 4. HCCS examples.

$\theta \wedge B_1 \wedge \cdots \wedge B_n \dashrightarrow H$ is *conjunctive* if θ is a conjunction of literals. We say that an HCCS $\mathcal{H}$ is conjunctive if each $hc \in \mathcal{H}$ is conjunctive.

We say that $\mathcal{H}$ is *recursion-free* if $\rightsquigarrow_{\mathcal{H}}$ is acyclic. We say that a recursion-free HCCS $\mathcal{H}$ is *tree-like* [12, 13] if 1.) there is exactly one root clause in $\mathcal{H}$ and every $P \in pvs(\mathcal{H})$ can reach a predicate variable occurring in the body of the root clause via $\rightsquigarrow_{\mathcal{H}}^*$; and 2.) for any $P \in pvs(\mathcal{H})$, at most one $hc \in \mathcal{H}$ contains P in its body, at most one $hc \in \mathcal{H}$ contains P as its head, and no $hc \in \mathcal{H}$ has multiple occurrences of P. For a tree-like HCCS $\mathcal{H}$, we define the *depth* of $\mathcal{H}$, $depth(\mathcal{H})$, to be the length of the longest $\rightsquigarrow_{\mathcal{H}}$ path. For η a mapping from predicate variables to predicate variables, we write $\eta(hc)$ for the Horn clause hc with each predicate variable application $P(\bar{x})$ replaced by $\eta(P)(\bar{x})$. We write $\eta(\mathcal{H})$ for $\{\eta(hc) \mid hc \in \mathcal{H}\}$. We say that a tree-like HCCS $\mathcal{H}'$ is an *unwound instance* of a (possibly recursive) HCCS $\mathcal{H}$ if there exists a mapping η from $pvs(\mathcal{H}')$ to $pvs(\mathcal{H})$ such that $\eta(\mathcal{H}') \subseteq \mathcal{H}$.

Constraint Solutions and Restricted Constraint Solvers. For σ a mapping from predicate variables to predicates in $\mathcal{T}$, we write $\sigma(hc)$ for hc with each predicate variable application $P(\bar{x})$ replaced by $\theta[\bar{x}/\bar{y}]$ where $\sigma(P) = \lambda\bar{y}.\theta$. We say that the map σ from $pvs(\mathcal{H})$ to predicates in $\mathcal{T}$ is a *solution* of $\mathcal{H}$, written $\sigma \vDash \mathcal{H}$, if for each $hc \in \mathcal{H}$, $\vDash \sigma(hc)$, interpreting $\dashrightarrow$ as $\Rightarrow$. We define $ran(\sigma)$, the *range* of σ, to be the set of predicates $\{\sigma(P) \mid P \in dom(\sigma)\}$.

We focus on constraint solving algorithms for tree-like HCCSs (they can be extended to arbitrary recursion-free HCCSs by adopting the technique from [13]). We say that an algorithm is an *unrestricted constraint solver* if given a tree-like HCCS $\mathcal{H}$, it returns a solution of $\mathcal{H}$ or decides that $\mathcal{H}$ has no solution. We say that an algorithm is an *exact L-restricted constraint solver* if given a tree-like HCCS $\mathcal{H}$, it decides if there is a solution σ of $\mathcal{H}$ such that $ran(\sigma) \subseteq L$ and returns such a solution if so. We say that an algorithm is a *relaxed (L_{base}, L_{ext})-restricted constraint solver* if given a tree-like HCCS $\mathcal{H}$, it either returns a solution σ of $\mathcal{H}$ such that $ran(\sigma) \subseteq L_{base} \cup L_{ext}$ or returns noproof indicating that it has failed to find a solution, with the requirement that it returns some solution (whose range is in $L_{base} \cup L_{ext}$) if there exists a solution σ' of $\mathcal{H}$ such that $ran(\sigma') \subseteq L_{base}$.

Example 3. Consider the HCCS $\mathcal{H}_{ex}$ shown in Figure 4. Here, $\bar{x} = a, b, x, y, z$, $\bar{x}' = a', b', x', y', z'$, and

$$
\begin{aligned}
\theta_{\mathsf{p1}} &\equiv x = a \wedge y = b \wedge z = 0 \\
\theta_{\mathsf{p2}} &\equiv z' = z + 1 \wedge y' = y + 1 \wedge x' = x \wedge a' = a \wedge b' = b \\
\theta_{\mathsf{p3}} &\equiv z \neq 0 \wedge z' = z - 1 \wedge y' = y - 1 \wedge x' = x \wedge a' = a \wedge b' = b \\
\theta_{\mathsf{p4}} &\equiv z = 0 \wedge a = b \wedge x \neq y
\end{aligned}
$$

$\mathcal{H}_{ex}$ is not tree-like (in fact, $\leadsto_S$ is cyclic). Figure 4 shows HCCSs $\mathcal{H}_{exa}$ and $\mathcal{H}_{exb}$ that are tree-like. In addition, they are unwound instances of $\mathcal{H}_{ex}$ because $\eta_a(\mathcal{H}_{exa}) \subseteq \mathcal{H}_{ex}$ and $\eta_b(\mathcal{H}_{exb}) \subseteq \mathcal{H}_{ex}$ where $\eta_a = \{P_1 \mapsto P, P_2 \mapsto P, Q_1 \mapsto Q, Q_2 \mapsto Q\}$ and $\eta_b = \{P_1 \mapsto P, P_2 \mapsto P, P_3 \mapsto P, Q_1 \mapsto Q, Q_2 \mapsto Q, Q_3 \mapsto Q\}$.

Recall the predicates $\theta_{ext1}, \theta_0, \theta_1, \theta_2$ from Example 1. Let the maps $\sigma_{a_1}, \sigma_{a_2}$, σ_{b_1}, and σ_{b_2} be defined as below.

$$
\sigma_{a_1} = \{P \mapsto \theta_{ex1} \mid P \in pvs(\mathcal{H}_{exa})\} \qquad \sigma_{b_1} = \{P \mapsto \theta_{ex1} \mid P \in pvs(\mathcal{H}_{exb})\}
$$

$$
\begin{aligned}
\sigma_{a_2} &= \{P \mapsto \theta_0 \mid P \in \{P_1, Q_2\}\} & \sigma_{b_2} &= \{P \mapsto \theta_0 \mid P \in \{P_1, Q_3\}\} \\
&\cup \{P \mapsto \theta_1 \mid P \in \{P_2, Q_1\}\} & &\cup \{P \mapsto \theta_1 \mid P \in \{P_2, Q_2\}\} \\
& & &\cup \{P \mapsto \theta_2 \mid P \in \{P_3, Q_1\}\}
\end{aligned}
$$

Then, σ_{a_1} and σ_{a_2} are solutions of $\mathcal{H}_{exa}$, and σ_{b_1} and σ_{b_2} are solutions of $\mathcal{H}_{exb}$.
▲

Relating Refinement Process to Constraint Solving. We relate constraint solving to refinement process. Roughly, the relationship says that, for any counterexample, there is a corresponding tree-like HCCS such that the range of its solutions are the proofs of the counterexample's spuriousness. We further assume that such a tree-like HCCS is always an unwound instance of some fixed "generator" HCCS determined by the given program.

We formalize the relationship. Let M be a program. We assume that there exists an HCCS $\mathcal{H}_{gen(M)}$ such that for any counterexample M' of M (i.e., $\mathsf{Abs}(M, F) = \mathsf{cex}(M')$ for some F), there exists an unwound instance $\mathcal{H}_{M'}$ of $\mathcal{H}_{gen(M)}$ that satisfies the following:

- if $\sigma \vDash \mathcal{H}_{M'}$ then $\mathsf{Abs}(M', ran(\sigma)) = \mathsf{safe}$ (i.e., the range of a solution of $\mathcal{H}_{M'}$ is a proof of M''s spuriousness); and
- if $\mathsf{Abs}(M', F) = \mathsf{safe}$ then $\exists \sigma. ran(\sigma) \subseteq F \wedge \sigma \vDash \mathcal{H}_{M'}$ (i.e., if M' can be refuted by F, then there is a solution for $\mathcal{H}_{M'}$ whose range is in F).

Hence, the task of implementing a relaxed language restricted refinement process RlxRef for the restricting language (L_{base}, L_{ext}) is now reduced to implementing a relaxed (L_{base}, L_{ext})-restricted constraint solver.

We remark that the relationship stated above is quite general and many CEGAR-based verifiers [17, 8, 2, 1, 13, 12, 21] use the relationship to implement the refinement process as a constraint solver for tree-like HCCSs. For example, refuting a counterexample in a typical CEGAR-based verification of sequential

imperative programs is equivalent to solving a tree-like HCCS of the form below where $\bar{x}$ are the variables in the program, and each θ_i is a formula on $\bar{x}$ and $\bar{x}'$ that expresses the semantics of symbolically executing the corresponding segment (e.g., basic block) in the path:

$$
\begin{array}{ll}
\theta_1 \rightarrow P_1(\bar{x}) & \theta_i \wedge P_i(\bar{x}) \rightarrow P_{i+1}(\bar{x}') \\
\theta_2 \wedge P_1(\bar{x}) \rightarrow P_2(\bar{x}') & \qquad \vdots \\
\qquad \vdots & P_n(\bar{x}) \rightarrow \bot
\end{array}
$$

In such a verification, the generator HCCS $\mathcal{H}_{gen(M)}$ can be described as follows. Let $\bar{x}$ be the variables in the program. For each node a in the program's control flow graph (CFG), we associate a predicate variable P_a of arity $|\bar{x}|$. For each edge from node a to node b in the CFG, we add to $\mathcal{H}_{gen(M)}$ the Horn clause $\theta_{ab} \wedge P_a(\bar{x}) \rightarrow P_b(\bar{x}')$ where θ_{ab} is a formula on $\bar{x}$ and $\bar{x}'$ expressing the effect of symbolically executing the CFG path from a to b (with $\bar{x}$ representing the current and $\bar{x}'$ representing the post state). For the entry node a, we add the Horn clause $\theta_{init} \rightarrow P_a(\bar{x})$ where θ_{init} is a formula on $\bar{x}$ expressing the program's initial state. Finally, for each error node a (i.e., `assert false` statement), we add the Horn clause $P_a(\bar{x}) \rightarrow \bot$.

Example 4. Recall the program M_{ex} from Example 1. The corresponding generator HCCS $\mathcal{H}_{gen(M_{ex})}$ is $\mathcal{H}_{ex}$ from Example 3. Roughly, the predicate variable P in the HCCS represents the program states at the time when the first loop is entered, and Q represents the states when the second loop is entered.

Recall the counterexamples M_{exa} and M_{exb} from Example 1, and the tree-like HCCSs $\mathcal{H}_{exa}$ and $\mathcal{H}_{exb}$ from Example 3. $\mathcal{H}_{exa}$ corresponds to M_{exa} and $\mathcal{H}_{exb}$ corresponds to M_{exb}. Indeed, as shown in Example 1, $\{\theta_{ex1}\}$ (resp. $\{\theta_0, \theta_1\}$) is a proof of M_{exa}, and $\mathcal{H}_{exa}$ has the corresponding solution σ_{a_1} (resp. σ_{a_2}) from Example 3. Similarly, the solutions σ_{b_1} and σ_{b_2} of $\mathcal{H}_{exb}$ and the proofs $\{\theta_{ex1}\}$ and $\{\theta_0, \theta_1, \theta_2\}$ of M_{exb} correspond. ▲

3.2 The Constraint Solver RlxSolveA

RlxSolveA is a relaxed $(L_{base}, L_{base}{}^{\wedge\vee})$-restricted constraint solver. It is parameterized by an exact L_{base}-restricted constraint solver that it takes as module. Let us fix the exact solver, $\mathcal{AE}_{L_{base}}$, and write RlxSolveA$[\mathcal{AE}_{L_{base}}]$ for RlxSolveA parameterized by the exact solver. Note that $L_{base}{}^{\wedge\vee}$ is finite for a finite L_{base}.

We briefly overview the construction of RlxSolveA. First, we leverage the equivalence of solving tree-like HCCS and tree interpolation [13] to reduce the problem to tree interpolation. Then, we adopt the standard proof-based interpolation technique that obtains interpolants from resolution proofs generated via SMT solving [10], except that we modify the SMT solver to use the exact L_{base}-restricted solver $\mathcal{AE}_{L_{base}}$ for the theory solver so as to infer L_{base}-restricted (partial) interpolants at the theory level of the resolution proof. As we shall show, this guarantees that if the SMT solver fails to prove, then no L_{base} solution exists, and conversely, any inferred solution is guaranteed to be in $L_{base}{}^{\wedge\vee}$.

We describe the approach more formally. First, we review *tree interpolation*. The tree interpolation problem takes as input (V, E, Θ) where (V, E) is a finite directed tree with the node set V and $(v, v') \in E$ denoting that the node v is a direct child of the node v', and the map Θ labels each node $v \in V$ with the $\mathcal{T}$-formula $\Theta(v)$. The goal is to find a map I from V to $\mathcal{T}$-formulas, called a *tree interpolant* of (V, E, Θ), that satisfies the following.

- $I(v_{rt}) = \bot$ for the root node v_{rt};
- for each $v \in V$, $\vDash \Theta(v) \wedge \bigwedge_{(v',v) \in E} I(v') \Rightarrow I(v)$; and
- for each $v \in V$, $fvs(I(v)) \subseteq (\bigcup_{(v',v) \in E^*} fvs(\Theta(v'))) \cap (\bigcup_{(v',v) \notin E^*} fvs(\Theta(v')))$.

We reduce constraint solving for a tree-like HCCS to tree interpolation as follows.[8] Let $\mathcal{H}$ be the input tree-like HCCS. We transform $\mathcal{H}$ to an equivalent HCCS that satisfies: 1.) for each predicate variable $P \in pvs(\mathcal{H})$, there exists a vector of fresh variables $\bar{x}_P$ such that P only occurs in the form $P(\bar{x}_P)$, and 2.) the only sharing of variables among Horn clauses are $\bar{x}_P$'s between two Horn clauses both containing P. Then, the transformed $\mathcal{H}$ is reduced to the tree interpolation problem $(V_{\mathcal{H}}, E_{\mathcal{H}}, \Theta_{\mathcal{H}})$ where

- $V_{\mathcal{H}} = pvs(\mathcal{H}) \cup \{v_{rt}\}$ where $v_{rt} \notin pvs(\mathcal{H})$;
- $E_{\mathcal{H}} = \leadsto_{\mathcal{H}} \cup \{(P, v_{rt}) \mid \theta \wedge \ldots P(\bar{x}_P) \ldots \dashrightarrow \bot \in \mathcal{H}\}$;
- For each P, $\Theta_{\mathcal{H}}(P) = \theta_P$ if $\theta_P \wedge \bigwedge_i B_i \dashrightarrow P(\bar{x}_P) \in \mathcal{H}$ and otherwise $\Theta_{\mathcal{H}}(P) = \bot$; and
- $\Theta_{\mathcal{H}}(v_{rt}) = \theta_{rt}$ where $\theta_{rt} \wedge \bigwedge_i B_i \dashrightarrow \bot \in \mathcal{H}$.

The theorem below follows from the construction, and shows the one-to-one correspondence between the tree interpolants of $(V_{\mathcal{H}}, E_{\mathcal{H}}, \Theta_{\mathcal{H}})$ and the solutions of $\mathcal{H}$.

Theorem 3 ([13]). *Let $\mathcal{H}$ be a tree-like HCCS. Let σ and I be such that $I(v_{rt}) = \bot$ and for each $P \in pvs(S)$, $\sigma(P) = \lambda \bar{x}_P.I(P)$. Then, $\sigma \vDash \mathcal{H}$ if and only if I is a tree interpolant of $(V_{\mathcal{H}}, E_{\mathcal{H}}, \Theta_{\mathcal{H}})$.*

Example 5. Recall the tree-like HCCS $\mathcal{H}_{exa}$ from Example 3. The corresponding tree interpolation problem (V, E, Θ) is shown below where each $\bar{x}_{P_1}$, $\bar{x}_{P_2}$, $\bar{x}_{Q_1}$, $\bar{x}_{Q_2}$ is a quintuple of fresh variables.

$$
\begin{aligned}
V &= \{v_{rt}, P_1, P_2, Q_1, Q_2\} \\
E &= \{(P_1, P_2), (P_2, Q_1), (Q_1, Q_2), (Q_2, v_{rt})\} \\
\Theta(P_1) &= \theta_{\mathsf{p1}}[\bar{x}_{P_1}/\bar{x}] \qquad \Theta(P_2) = \theta_{\mathsf{p2}}[\bar{x}_{P_1}/\bar{x}][\bar{x}_{P_2}/\bar{x}'] \\
\Theta(Q_1) &= \bar{x}_{P_2} = \bar{x}_{Q_1} \qquad \Theta(Q_2) = \theta_{\mathsf{p3}}[\bar{x}_{Q_1}/\bar{x}][\bar{x}_{Q_2}/\bar{x}'] \\
\Theta(v_{rt}) &= \theta_{\mathsf{p4}}[\bar{x}_{Q_2}/\bar{x}]
\end{aligned}
$$

▲

Now, the relaxed $(L_{base}, L_{base}{}^{\wedge\vee})$-restricted constraint solving problem is reduced to relaxed $(L_{base}, L_{base}{}^{\wedge\vee})$-restricted tree interpolation. That is, we would like to find tree interpolants restricted to $L_{base} \cup L_{base}{}^{\wedge\vee}$ (i.e., $L_{base}{}^{\wedge\vee}$), with the guarantee to return one if there exists a L_{base}-restricted tree interpolant.

---

[8] The reduction is adopted from [13].

$$\text{THY} \quad \frac{\mathcal{AE}_{L_{base}}(\mathcal{H}_{thy(C,V,E,\Theta)}) = \sigma \quad I(v_{rt}) = \bot \quad \forall P.\lambda\bar{x}_P.I(P) = \sigma(P)}{(V,E,\Theta) \vdash_{itp} C : I}$$

$$\text{HYP} \quad \frac{C \in \Theta(v) \quad \forall v'.I(v') = \begin{cases} C\!\uparrow_{v'} & \text{if } (v,v') \in E^* \\ \top & \text{otherwise} \end{cases}}{(V,E,\Theta) \vdash_{itp} C : I}$$

$$\text{RES} \quad \frac{(V,E,\Theta) \vdash_{itp} p \vee C_1 : I_1 \qquad \forall v.I_3(v) = \begin{cases} I_1(v) \wedge I_2(v) & \text{if } p \in outs(v) \\ I_1(v) \vee I_2(v) & \text{otherwise} \end{cases}}{(V,E,\Theta) \vdash_{itp} \neg p \vee C_2 : I_2}}{(V,E,\Theta) \vdash_{itp} C_1 \vee C_2 : I_3}$$

Fig. 5. The tree interpolation rules.

Next, we describe the process of relaxed $(L_{base}, L_{base}{}^{\wedge\vee})$-restricted tree inter-polation. In what follows, we assume familiarity with lazy SMT and the proof-based technique for obtaining interpolants from resolution proofs [16, 10]. Let (V, E, Θ) be the tree interpolation instance to be solved. In an ordinary proof-based tree interpolation, one looks for tree interpolants by having the SMT solver check the unsatisfiability of $\bigwedge_{v \in V} \Theta(v)$ and analyzing the output resolution proof to compute the interpolant. However, this decides the existence of, and infers, a tree interpolant from the entire $\mathcal{T}$, and is unsuitable for our task (i.e., this results in an unrestricted constraint solver).

Instead, we modify the SMT solver so that its theory-level reasoning is del-egated to the exact L_{base}-restricted constraint solver $\mathcal{AE}_{L_{base}}$. More specifically, when the SMT solver builds a model of possible (propositional) satisfying assign-ment $\neg C$, instead of passing the model to a theory solver as in ordinary SMT, we build a "fragment" HCCS $\mathcal{H}_{thy(C,V,E,\Theta)}$ that just contains the part of the tree interpolation problem touched by the literals in C. Formally,

$$\mathcal{H}_{thy(C,V,E,\Theta)} = \{\neg C\!\downarrow_v \wedge \bigwedge_{(P,v) \in E} P(\bar{x}_P) \rightarrow H_v \mid v \in V\}$$

where $C\!\downarrow_v$ is the set of literals of C over atomic predicates occurring in $\Theta(v)$, and $H_v = \bot$ if $v = v_{rt}$ and $H_v = Q(\bar{x}_Q)$ if v is a predicate variable Q. We pass $\mathcal{H}_{thy(C,V,E,\Theta)}$ to $\mathcal{AE}_{L_{base}}$ to decide if it has a L_{base} restricted solution, and if so, we set the obtained solution as the partial tree interpolant for the theory-level resolution proof nodes where C occurs as the theory lemma. Otherwise, we can safely reject that the whole problem as having no L_{base}-restricted tree interpolant and return noproof. To generate the tree interpolant for the whole, we adopt the proof-based approach that builds the tree interpolant in a bottom up manner following the rules shown in Figure 5.[9] Here, $outs(v)$ is the set of atomic predicates occurring outside of the subtree rooted at v (i.e., $outs(v) =$

---

[9] We assume that each $\Theta(v)$ is CNF (if not, they can be transformed so via the Tseitin transformation [19]).

$\{p \mid p \text{ occurs in } \Theta(v') \text{ where } (v', v) \notin E^*\})$, and $C\uparrow_v$ is the set of literals of C over the atomic predicates occurring outside of the subtree rooted at v (i.e., $C\uparrow_v = \{p \in C \mid p \in outs(v)\} \cup \{\neg p \in C \mid p \in outs(v)\})$. The rules HYP for clauses in the input tree and RES for resolution steps extend the analogous rules from the standard proof-based interpolation [10] to tree interpolation. As described above, THY uses the L_{base}-restricted solution computed by the exact solver for the partial tree interpolant. As we show in Theorems 4 and 5 below, this achieves the desired relaxed $(L_{base}, L_{base}{}^{\wedge\vee})$-restricted tree interpolation.

First, we show that any tree interpolant obtained by the method is restricted to $L_{base}{}^{\wedge\vee}$.

Theorem 4. *Let $\mathcal{H}$ be a tree-like HCCS and (V, E, Θ) be the corresponding tree interpolation problem. Suppose $(V, E, \Theta) \vdash_{itp} \bot : I$. Then, I is a tree interpolant of (V, E, Θ), and for all $P \in pvs(\mathcal{H})$, $\lambda \bar{x}_P.I(P) \in L_{base}{}^{\wedge\vee}$.*

Next, we prove that if there is a L_{base}-restricted tree interpolant for the given tree interpolation instance, then the method infers some tree interpolant (and by Theorem 4 above, such a tree interpolant will be restricted to $L_{base}{}^{\wedge\vee}$).

Theorem 5. *Let $\mathcal{H}$ be a tree-like HCCS and (V, E, Θ) be the corresponding tree interpolation problem. Suppose there is a tree interpolant I of (V, E, Θ) such that for all $P \in pvs(\mathcal{H})$, $\lambda \bar{x}_P.I(P) \in L_{base}$. Then, $(V, E, \Theta) \vdash_{itp} \bot : I'$ for some I'.*

We note that $\mathcal{H}_{thy(C, V, E, \Theta)}$ is always a conjunctive HCCS and is often much smaller than the input HCCS. Therefore, the expensive exact L_{base}-restricted constraint solver $\mathcal{AE}_{L_{base}}$ is only applied to small conjunctive HCCSs, thereby making its job easier. Also, we note that, while we have presented RlxSolveA to be parameterized by an exact L_{base}-restricted constraint solver passed as module, the algorithm actually works even if a relaxed (L_{base}, L_{ext})-restricted constraint solver is used in place of the exact L_{base}-restricted constraint solver.[10] Therefore, RlxSolveA can actually be parameterized by RlxSolveA itself, but the solvers must be "primed" by some exact solver (e.g., RlxSolveA[RlxSolveA[$\mathcal{AE}_{L_{base}}$]]).

Minimizing Theory Lemmas. When $\neg C$ is given as a possible propositional model by the SMT solver, we use the exact finite-language-restricted constraint solver $\mathcal{AE}_{L_{base}}$ to find a solution for $\mathcal{H}_{thy(C, V, E, \Theta)}$. But, using C directly as the theory lemma after $\mathcal{AE}_{L_{base}}$ finds a solution could result in the SMT solver producing many propositional models and lead to bad performance. (This is analogous to using C directly as the theory lemma in an ordinary lazy SMT solving when the theory solver finds $\neg C$ unsatisfiable.) Instead, we let $\mathcal{AE}_{L_{base}}$ return the subset of the literals of C that it used to find the solution, and use it to obtain a smaller theory lemma. (We refer to the extended report [18] for more detail.)

---

[10] More precisely, it becomes a relaxed $(L_{base}, L_{ext}{}^{\wedge\vee})$-restricted constraint solver when passed a relaxed (L_{base}, L_{ext})-restricted constraint solver.

3.3 The Constraint Solver RlxSolveB

RlxSolveB is a relaxed $(L_{base}, \mathsf{LB}(L_{base} \cup L_{ext}, \mathcal{AU}, \ell))$-restricted constraint solver which takes as module a relaxed (L_{base}, L_{ext})-restricted constraint solver, an unrestricted constraint solver $\mathcal{AU}$, and a positive integer parameter ℓ. $\mathsf{LB}(L, \mathcal{AU}, \ell)$ is a finite language of predicates determined by L, $\mathcal{AU}$, and ℓ.

We informally describe RlxSolveB. We select some fraction of predicate variables in a certain "fair" manner based on the parameter ℓ and use the relaxed (L_{base}, L_{ext})-restricted solver provided as a module to look for solutions to just the selected predicate variables. After restricted solutions are obtained for the selected predicate variables, we use $\mathcal{AU}$ to look for unrestricted solutions to the remaining predicate variables, and return the combined solution as the solution for the input HCCS. Note that this technique reduces the number of predicate variables that the given relaxed (L_{base}, L_{ext})-restricted solver needs to solve for, and therefore can be used to improve the performance of a relaxed (L_{base}, L_{ext})-restricted solver. The key observation we use here is that HCCSs solved in a refinement process are all unwound instances of a fixed "generator" HCCS (i.e., $\mathcal{H}_{gen(M)}$). As we shall show next, the observation can be used to guarantee that the result is a relaxed finite-language-restricted solver, when the predicate variable selection is done in a certain proper way.

We describe the constraint solving algorithm in detail. In what follows, we extend the definition of a tree-like HCCS (cf. Section 3.1) so that the root clause can be a Horn clause whose head is of the form $P(\bar{x})$ where P does not occur anywhere else in the HCCS. For such an HCCS $\mathcal{H}$, we say that P is the *root* of $\mathcal{H}$ and write $root(\mathcal{H}) = P$ ($root(\mathcal{H}) = \perp$ for $\mathcal{H}$ with a $\perp$-head root clause).

```
01: RlxSolveB[AR_{(L_base,L_ext)}, AU, ℓ](H) =
02:   let Y = partition(ℓ, H) in
03:   let A = ∪_{H'∈Y} {root(H')} ∖ {⊥} in
04:   let H_A = rewrite(H, A) in
05:   match AR_{(L_base,L_ext)}(H_A) with
06:       noproof → return noproof
07:     | sol(σ_A) → return sol(σ_A ∪ ∪_{H'∈Y} AU(σ_A(H')))
```

Fig. 6. The overview of RlxSolveB.

Let RlxSolveB be parameterized by the relaxed (L_{base}, L_{ext})-restricted constraint solver $\mathcal{AR}_{(L_{base}, L_{ext})}$, the unrestricted constraint solver $\mathcal{AU}$, and the positive integer ℓ. Figure 6 shows the overview of RlxSolveB. RlxSolveB first partitions the input HCCS $\mathcal{H}$ into a set of tree-like HCCSs of depth at most ℓ in a top-down manner (line 2). Formally, partition is defined as follows.

$$
\begin{aligned}
&\mathsf{partition}(\ell, \mathcal{H}) = \\
&\quad \textbf{if } depth(\mathcal{H}) \leq \ell \textbf{ then } \{\mathcal{H}\} \\
&\quad \textbf{else let } \mathcal{H}', X = subtrees(\ell, \mathcal{H}) \textbf{ in} \\
&\quad\quad \{\mathcal{H}'\} \cup \bigcup_{\mathcal{H} \in X} \mathsf{partition}(\ell, \mathcal{H})
\end{aligned}
$$

Here, $subtrees(\ell, \mathcal{H})$ returns the pair $(\mathcal{H}', X)$ such that 1.) $\mathcal{H}'$ is the largest tree-like subset of $\mathcal{H}$ containing the root clause of $\mathcal{H}$ and $depth(\mathcal{H}') = \ell$, and 2.) X is the set of subtrees of $\mathcal{H}$ rooted at each leaf of $\mathcal{H}'$. It is easy to see that Y partitions $\mathcal{H}$ (i.e., $\mathcal{H} = \bigcup Y$ and $\forall \mathcal{H}_1, \mathcal{H}_2 \in Y.\mathcal{H}_1 \neq \mathcal{H}_2 \Rightarrow \mathcal{H}_1 \cap \mathcal{H}_2 = \varnothing$), and that each $\mathcal{H}' \in Y$ is a tree-like HCCS of depth at most ℓ. In fact, the partition is the coarsest of such partitions, where the only HCCSs in the partition having depth less than ℓ are the ones whose leaf predicate variables do not appear anywhere else in the partition.

By construction, only the root predicate variables are shared by different HCCSs in Y (more precisely, a root of one HCCS appears as a leaf in another HCCS). RlxSolveB selects these shared predicate variables to be the ones to infer restricted solutions (line 3). It can be seen that the fraction of the selected predicate variables, that is $|A|/|pvs(\mathcal{H})|$, is inversely proportional to the size of a depth ℓ tree-like subset of $\mathcal{H}$, and decreases rapidly as ℓ is increased.

To infer restricted solutions to A, RlxSolveB constructs the HCCS $\mathcal{H}_A$ such that $pvs(\mathcal{H}_A) = A$, and solutions of $\mathcal{H}_A$ correspond exactly to the solutions of $\mathcal{H}$ restricted to A (i.e., $\sigma \vDash \mathcal{H}_A$ if and only if $\exists \sigma'. \sigma' \restriction_A = \sigma \wedge \sigma' \vDash \mathcal{H}$). This is done by the operation $\mathsf{rewrite}(\mathcal{H}, A)$ (line4), defined to be the application of the following rewriting relation $\twoheadrightarrow$ to $\mathcal{H}$ until convergence.

$$\mathcal{H}' \cup \{\Phi_1 \dashrightarrow P(\bar{x}), \Phi_2 \wedge P(\bar{y}) \dashrightarrow H\} \twoheadrightarrow \mathcal{H}' \cup \{\Phi_2 \wedge \Phi_1[\bar{y}/\bar{x}] \dashrightarrow H\}$$

Here, $P \in pvs(\mathcal{H}) \smallsetminus A$, and Φ_i's range over Horn clause bodies. (We assume that each Horn clause in $\mathcal{H}$ is over disjoint variables. Otherwise, we transform $\mathcal{H}$ into such a form by variable renaming.)

Then, RlxSolveB calls $\mathcal{AR}_{(L_{base}, L_{ext})}$ to find a $L_{base} \cup L_{ext}$-restricted solution for $\mathcal{H}_A$ (line 5). If $\mathcal{AR}_{(L_{base}, L_{ext})}$ returns noproof then no L_{base} solution exists for $\mathcal{H}_A$ by the property of $\mathcal{AR}_{(L_{base}, L_{ext})}$, and by the construction above, it can be shown that no L_{base} solution exists for the input HCCS $\mathcal{H}$ either, and we safely return noproof (line 6) (see Theorem 7 for the proof). Otherwise, we obtain a solution σ_A for $\mathcal{H}_A$, and RlxSolveB calls $\mathcal{AU}$ on each element of the partition with the solution σ_A substituted (i.e., $\mathcal{AU}(\sigma_A(\mathcal{H}'))$ for each $\mathcal{H}' \in Y$).[11] This gives solutions for the remaining predicate variables in $\mathcal{H}$, and we return the union of σ_A and these solutions as the final solution (line 7).

We argue that the produced solution is indeed a solution of $\mathcal{H}$. Let $Y = \{\mathcal{H}_1, \ldots, \mathcal{H}_n\}$. Note that the only predicate variables shared by different elements in Y are A. Therefore, $\sigma_A(\mathcal{H}_1), \ldots, \sigma_A(\mathcal{H}_n)$, and $\mathcal{H}_A$ are over disjoint predicate variables, and their solutions are over disjoint domains. Then, because Y partitions $\mathcal{H}$, it follows that $\sigma_A \cup \bigcup_{\mathcal{H}' \in Y} \mathcal{AU}(\sigma_A(\mathcal{H}'))$ is a solution of $\mathcal{H}$.

To show that RlxSolveB is a relaxed finite-language-restricted constraint solver, it remains to show that the obtained solution is restricted to a finite language of predicates. We show that it is restricted to $\mathsf{LB}(L_{base} \cup L_{ext}, \mathcal{AU}, \ell)$ which is

---

[11] Here, we extend the notion of substitution so that the result is tree-like: for $P \in dom(\sigma_A)$, $\sigma_A(\Phi \dashrightarrow P(\bar{x})) = \neg \sigma_A(P)[\bar{x}/\check{\nu}(P)] \wedge \sigma_A(\Phi) \dashrightarrow \bot$.

defined as follows. Let L be a finite language. We define $\mathsf{LB}(L, \mathcal{AU}, \ell)$ as follows.

$$\mathsf{LB}(L, \mathcal{AU}, \ell) = L \cup \bigcup_{\mathcal{H}' \in X} ran(\mathcal{AU}(\mathcal{H}'))$$
where
$$X = \{\sigma(\mathcal{H}') \mid \mathcal{H}' \in unwds(\ell, \mathcal{H}_{gen(M)}) \text{ and } \sigma \succeq_L \mathcal{H}'\}$$

Here, M is the program being verified (i.e., the input to the top-level procedure RlxCegar), $unwds(\ell, \mathcal{H})$ is the set of unwound instances of $\mathcal{H}$ of depth at most ℓ, and $\sigma \succeq_L \mathcal{H}$ if and only if σ is a map from the leaves and the root of $\mathcal{H}$ to the predicates in L (i.e., $dom(\sigma) = leaves(\mathcal{H}) \cup \{root(\mathcal{H})\} \setminus \{\bot\}$ and $ran(\mathcal{H}) \subseteq L$). Note that $\mathcal{H}_{gen(M)}$ is a constant for the entire run of RlxCegar(M), and $unwds(\ell, \mathcal{H})$ is finite for any $\mathcal{H}$ and ℓ. Therefore, $\mathsf{LB}(L, \mathcal{AU}, \ell)$ is a finite language that is determined by L, $\mathcal{AU}$ and ℓ.

We formally prove that RlxSolveB is indeed a relaxed $(L_{base}, \mathsf{LB}(L_{base} \cup L_{ext}, \mathcal{AU}, \ell))$-restricted constraint solver. First, we prove that the solution returned is indeed a solution of the input HCCS and that it is restricted to $\mathsf{LB}(L_{base} \cup L_{ext}, \mathcal{AU}, \ell)$.

Theorem 6. *Suppose RlxSolveB$[\mathcal{AR}_{(L_{base}, L_{ext})}, \mathcal{AU}, \ell](\mathcal{H})$ returns sol(σ). Then, $\sigma \models \mathcal{H}$ and $ran(\sigma) \subseteq \mathsf{LB}(L_{base} \cup L_{ext}, \mathcal{AU}, \ell)$.*

Next, we show that some solution is returned if there exists a L_{base}-restricted solution to the given HCCS (and by Theorem 6, such a solution is restricted to $\mathsf{LB}(L_{base} \cup L_{ext}, \mathcal{AU}, \ell)$).

Theorem 7. *Suppose that there exists σ such that $\sigma \models \mathcal{H}$ and $ran(\sigma) \subseteq L_{base}$. Then, RlxSolveB$[\mathcal{AR}_{(L_{base}, L_{ext})}, \mathcal{AU}, \ell](\mathcal{H})$ infers some σ' such that $\sigma' \models \mathcal{H}$.*

Example 6. Recall the HCCS $\mathcal{H}_{exa}$ from Example 3. Running RlxSolveB on $\mathcal{H}_{exa}$ with $\ell = 2$, we have $Y = \mathsf{partition}(2, \mathcal{H}_{exa}) = \{\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3\}$ where

$$\mathcal{H}_1 = \{\theta_{\mathsf{p3}} \wedge Q_1(\bar{x}) \dashrightarrow Q_2(\bar{x}'), \theta_{\mathsf{p4}} \wedge Q_2(\bar{x}) \dashrightarrow \bot\}$$
$$\mathcal{H}_2 = \{\theta_{\mathsf{p2}} \wedge P_1(\bar{x}) \dashrightarrow P_2(\bar{x}'), P_2(\bar{x}) \dashrightarrow Q_1(\bar{x})\}$$
$$\mathcal{H}_3 = \{\theta_{\mathsf{p1}} \dashrightarrow P_1(\bar{x})\}$$

Then, the shared predicate variables that are selected to be restricted are $A = \{root(\mathcal{H}_1), root(\mathcal{H}_2), root(\mathcal{H}_3)\} \setminus \{\bot\} = \{P_1, Q_1\}$. And, $\mathcal{H}_A = \mathsf{rewrite}(\mathcal{H}_{exa}, A)$ is as shown below.

$$\mathcal{H}_A = \{\theta_{\mathsf{p3}} \wedge Q_1(\bar{x}) \wedge \theta_{\mathsf{p4}}[\bar{x}', \bar{x}''/\bar{x}, \bar{x}'] \dashrightarrow \bot, \theta_{\mathsf{p2}} \wedge P_1(\bar{x}) \dashrightarrow Q_1(\bar{x}'), \theta_{\mathsf{p1}} \dashrightarrow P_1(\bar{x})\}$$

where $\bar{x}''$ is a quintuple of fresh variables. RlxSolveB then calls $\mathcal{AR}_{(L_{base}, L_{ext})}$ on $\mathcal{H}_A$ to obtain a restricted solution for A. Suppose the returned solution is $\sigma_A = \{P \mapsto a = b \Rightarrow y = z + x \mid P \in \{P_1, Q_1\}\}$. Then, σ_A is applied to each element of the partition and we obtain $\sigma_A(\mathcal{H}_1)$, $\sigma_A(\mathcal{H}_2)$, and $\sigma_A(\mathcal{H}_3)$ shown below.

$$\sigma_A(\mathcal{H}_1) = \{\theta_{\mathsf{p3}} \wedge (a = b \Rightarrow y = x + z) \dashrightarrow Q_2(\bar{x}'), \theta_{\mathsf{p4}} \wedge Q_2(\bar{x}) \dashrightarrow \bot\}$$
$$\sigma_A(\mathcal{H}_2) = \{\theta_{\mathsf{p2}} \wedge (a = b \Rightarrow y = x + z) \dashrightarrow P_2(\bar{x}'), a = b \wedge y \neq x + z \wedge P_2(\bar{x}) \dashrightarrow \bot\}$$
$$\sigma_A(\mathcal{H}_3) = \{a = b \wedge y \neq x + z \wedge \theta_{\mathsf{p1}} \dashrightarrow \bot\}$$

$\mathcal{AU}$ is called on $\sigma_A(\mathcal{H}_1)$, $\sigma_A(\mathcal{H}_2)$, and $\sigma_A(\mathcal{H}_3)$ to infer unrestricted solutions to the remaining predicate variables. Suppose we have obtained the solutions $\sigma_1 = \{Q_2 \mapsto a = b \Rightarrow z \neq -1 \wedge y = x + z\}$, $\sigma_2 = \{P_2 \mapsto a = b \Rightarrow y = x + z\}$, and $\sigma_3 = \varnothing$ for $\sigma_A(\mathcal{H}_1)$, $\sigma_A(\mathcal{H}_2)$, and $\sigma_A(\mathcal{H}_3)$ respectively. Finally, RlxSolveB returns the combined map, $\sigma_A \cup \sigma_1 \cup \sigma_2 \cup \sigma_3$, as the solution inferred for the input HCCS $\mathcal{H}_{exa}$. ▲

4 Implementation and Experiments

We have implemented the new refinement algorithms RlxSolveA and RlxSolveB described in Section 3. The refinement algorithms require an exact finite-language-restricted constraint solver and an unrestricted constraint solver to be provided as modules. An unrestricted constraint solver finds unrestricted solutions to the given tree-like HCCS. This is the ordinary constraint solving for tree-like HCCSs which is a well-studied problem, and we use the existing technique that iteratively solve the constraints one predicate variable at a time by using interpolation as a blackbox process (see [20, 4, 17] for details).[12]

Exact Finite-Language-Restricted Constraint Solver. For the exact solver, we use a simple approach in which the finite predicate languages are represented by predicate templates containing unknowns of bounded range. We use an SMT solver[13] to find an assignment to the unknowns within the bound that makes the templates into an actual solution. Below, we informally describe the process by an example, and defer the detailed description to the extended report [18].

Example 7. Recall the program M_{ex} from Example 1. Let L_{base} be the finite language of predicates consisting of conjunctions of at most two atomic predicates whose numeric constants are bounded in the range $\{-1, 0, 1\}$. We represent the language by the *bounded predicate template* shown below

$$\lambda a, b, x, y, z.\, c_1 a + c_2 b + c_3 x + c_4 y + c_5 z + c_6 \leq 0 \wedge$$
$$c_7 a + c_8 b + c_9 x + c_{10} y + c_{11} z + c_{12} \leq 0$$

where c_i's are *unknown constants* each associated with the *bound* $\{-1, 0, 1\}$. Bounded predicate templates can concisely represent a finite language of predicates.[14]

Let ξ be a bounded predicate template. To check if the given HCCS $\mathcal{H}$ has a solution in the language represented by ξ, we make a *solution template* σ_ξ that maps each predicate variable in $\mathcal{H}$ to a copy of ξ with fresh unknowns. Then, we check if there exists an assignment to the unknowns within the bounds that makes the solution template into an actual solution of $\mathcal{H}$, that is, we look for assignments to the unknowns within the bounds that satisfy $\forall hc \in \sigma_\xi(\mathcal{H}).\ \vDash hc$.

---

[12] The implementation uses MathSAT 5 (`http://mathsat.fbk.eu/`) for the backend interpolation process.

[13] The implementation uses Z3 (`http://z3.codeplex.com/`).

[14] Note that such a language is generally not closed under conjunctions or disjunctions.

For QFLRA, the latter can be done by applying the Motzkin's transposition theorem [15] to reduce the problem to the satisfiability problem for quantifier-free non-linear real arithmetic, and using an SMT solver to solve the resulting problem. ▲

We note that the exact finite-language-restricted constraint solving is a highly expensive process and using it directly solve the whole HCCS is prohibitive. Indeed, as we show in the experiments, the exact solver fails to scale even on relatively small constraint sets (see also the discussion in Section 5 and the complexity theoretic analysis in the extended report [18]).

Experiment Setup. We have experimented with the new refinement algorithms by using them in the refinement process of MoCHi [8]. MoCHi is a state-of-the-art software model checker for higher-order functional programs based on predicate abstraction, CEGAR, and higher-order-recursion-scheme (HORS) model checking. MoCHi verifies assertion safety of OCaml programs. A verifier for functional programs such as MoCHi is suited for experimenting with the new refinement algorithm because Horn-clause constraints generated in such a verifier often contain non-trivial tree-like structure. (Intuitively, this is because the constraints express the flow of data in the program, and data often flow in a complex way in a functional program, e.g., passed to and returned from recursive functions, captured in closures, etc.)

The new refinement algorithms RlxSolveA and RlxSolveB are parametric. For this experiment, we parameterize them as follows to obtain a single refinement algorithm:

$$\mathsf{RlxSolveB}[\mathsf{RlxSolveA}[\mathcal{AE}_{L_{base}}], \mathcal{AU}, 4]$$

Here, the exact L_{base}-restricted constraint solver $\mathcal{AE}_{L_{base}}$ and the unrestricted solver $\mathcal{AU}$ are the ones described above. That is, we use RlxSolveB parameterized to use as modules the relaxed $(L_{base}, (L_{base})^{\wedge\vee})$-restricted constraint solver RlxSolveA (itself parameterized by the exact L_{base}-restricted constraint solver $\mathcal{AE}_{L_{base}}$) and the unrestricted constraint solver $\mathcal{AU}$, and with the parameter $\ell = 4$. The strata of restricting predicate languages are built "dynamically" as the CEGAR iteration progresses, by starting from a small fixed (L_{base}^0, L_{ext}^0) and enlarging the current (L_{base}, L_{ext}) whenever the refinement algorithm returns noproof by using the unrestricted refinement process.[15]

We compare the new refinement algorithm with two other refinement methods: 1.) the ordinary (incomplete) unrestricted predicate search, and 2.) exact finite-language-restricted predicate search. The unrestricted predicate search algorithm is $\mathcal{AU}$, and the exact finite-language restricted predicate search algorithm is $\mathcal{AE}$. For $\mathcal{AE}$, we give (the L_{base} part of) the same restricting predicate language given to the new algorithm when solving the corresponding HCCS.

We have ran the three refinement algorithms on 318 HCCSs generated by running MoCHi on 139 programs, measuring the time spent in each run of the

---

[15] Formally, this is done by having a non-decreasing preorder of restricting predicate languages where the limit of any ω-chain is $\mathcal{T}$, and when noproof is returned, the language raised to the least one containing the predicate inferred by $\mathcal{AU}$.

refinement process. The benchmark programs are mostly taken from the previous work on MoCHi [8, 14, 22, 9]. To obtain the benchmark HCCS set, we ran MoCHi on each benchmark program with the new refinement algorithm until completion or timeout and recorded the HCCS given as the input to each run of the refinement process. We also compare the overall verification speed of MoCHi when using the three refinement algorithms. This is done by running MoCHi with each of the refinement algorithm on the 139 benchmark programs. We have run the experiments on a machine with 2.69 GHz i7-4600U processor with 16 GB of RAM, with the time limit of 100 seconds. The benchmark programs, the benchmark HCCSs and the experiment results data are available online [18].

Experiment Aim and Hypothesis. Because of the overhead from computing restricted proofs, we expect the individual refinement runs to be slower with the new refinement algorithm compared to an ordinary incomplete approach which only does unrestricted refinement, but faster than the more naïve complete approach that directly applies the exact finite-language-restricted proof search to the entire refinement problem. The main purpose of the experiment is to test this hypothesis. We also compare the overall verification speeds, but we do not expect a significant improvement on this aspect because of the inherent complexity of the verification problem. (For any sound and QFLRA-complete verifier, one can always find a program on which the verifier takes arbitrarily long time.)

Experiment Results and Analysis. Figure 7 shows the plots comparing the the run times of the new refinement algorithm (New Algorithm), the unrestricted refinement algorithm (Unrestricted), and the exact finite-language-restricted refinement algorithm (Exact) on each of the 318 benchmark HCCSs. As we have expected, the unrestricted refinement algorithm is the fastest of the three. The new algorithm performs quite competitively, however, and shows that it is able to achieve completeness with only a low overhead. Also, the plots show that the exact finite-language-restricted refinement algorithm is significantly slower, timing out on many instances that the other two algorithms were able to solve quickly.

Figure 8 shows the plots comparing the run times of the overall verification process on each of the 139 benchmark programs for each refinement algorithm. The plots show that there is no clear winner in this comparison and none of the three outperformed the others on all benchmarks (while the unrestricted refinement edged out in the number of instances solved within the time limit, it also timed out on some instances the complete methods were able to solve). This is due to the inherent undecidability of the program verification problem, and the fact that the speed of overall verification depends heavily on subtle heuristic choices made by MoCHi. Such issues are largely outside of the scope of this paper, but they give interesting insights into what would be the good heuristics to use with the new refinement algorithm. For instance, an interesting behavior we have observed is that the stratified approaches (New Algorithm and Exact) sometimes infer more useless predicates than the ordinary unrestricted refinement because a stratified approach needs to add predicates to raise the lan-
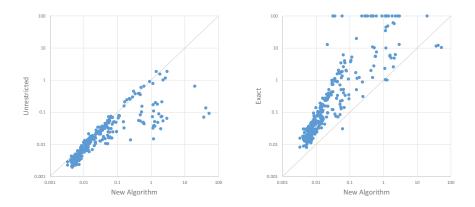
Fig. 7. Run time comparison of the refinement algorithms on benchmarks HCCSs
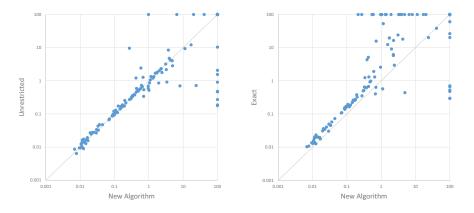

Fig. 8. Run time comparison of the refinement algorithms on benchmarks programs

guage stratum till it reaches the level where a proof of the given program exists. Because MoCHi does eager predicate abstraction, its performance degrades exponentially in the number of predicates that are added to the candidate predicate set. This seems to have had a large negative impact on the stratified approaches. A possible way to address the issue maybe is to have MoCHi take a more lazy approach to predicate abstraction, or allow the language strata "coarseness" to be dynamically adjustable so that we can immediately jump to a large predicate language when it seems beneficial.

5 Related Work

Previous work [7,11] has considered an *exact* stratification approach which requires the refinement process to exactly decide if a proof of the given counterex-

ample's spuriousness exists in the current finite language stratum. As remarked before, an issue with exact stratification is the high cost of exact finite-language-restricted proof search. As we have shown empirically in Section 4, the exact finite-language-restricted proof search suffers from high overhead. (We also show complexity theoretic evidences for the inherent hardness of the exact search in the extended report [18].) We note that relaxed stratification is a generalization of exact stratification. That is, exact stratification is a special case of relaxed stratification where $L_{ext} = \varnothing$.

We note that the interpolation technique that limits the theory-level reasoning to only emit restricted partial interpolants (cf. Section 3.2) has also been proposed in [7]. But, they target exact stratification and therefore requires the restricting language to be closed under conjunctions and disjunctions (so that $L^{\wedge\vee} = L$), which substantially reduces the applicability of the technique.[16]

6 Conclusion

We have presented a new approach to complete predicate refinement, called relaxed stratification, where the background theory is stratified into a sequence of finite predicate languages

$$(L_{base}^0, L_{ext}^0), (L_{base}^1, L_{ext}^1), \ldots (L_{base}^k, L_{ext}^k), \ldots$$

such that each run of the refinement process is restricted to only infer predicates from the current stratum $L_{base} \cup L_{ext}$. Contrary to previous approaches to complete refinement, the refinement process is neither required to decide the existence of a proof for the given counterexample in $L_{base} \cup L_{ext}$ nor in L_{base}, but is only required to return some proof if one exists in L_{base}. We have proved that the approach is complete despite the relaxed requirement, assuming that the strata of L_{base}'s grow to eventually cover the predicates of the underlying theory. We have shown that the relaxed requirement can be used to build practical refinement algorithms that have low overhead and the completeness guarantee.

References

1. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 405–416. ACM, 2012.
2. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In T. Ball and M. Sagiv, editors, *POPL*, pages 331–344. ACM, 2011.

---

[16] Contrary to [7], in QFLRA, it is insufficient to only look for an atomic interpolant (i.e., a separating hyperplane) when interpolating between even just conjunctions of literals (i.e., polytopes) under a finite-language restriction. For example, consider interpolating between $y \leq 1 \wedge 2x + y \leq -3 \wedge x \leq -1$ and $y + x \geq 1$ under the restriction that interpolants' constants are in $\{-1, 0, 1\}$.

3. A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In H. Yang, editor, *APLAS*, volume 7078 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2011.

4. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 471–482. ACM, 2010.

5. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In N. D. Jones and X. Leroy, editors, *POPL*, pages 232–244. ACM, 2004.

6. R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), 2009.

7. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.

8. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 222–233. ACM, 2011.

9. T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In Z. Shao, editor, *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 392–411. Springer, 2014.

10. K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

11. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2008.

12. P. Rümmer, H. Hojjat, and V. Kuncak. Classifying and solving horn clauses for verification. In E. Cohen and A. Rybalchenko, editors, *VSTTE*, volume 8164 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2013.

13. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2013.

14. R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In E. Albert and S. Mu, editors, *PEPM*, pages 53–62. ACM, 2013.

15. A. Schrijver. *Theory of linear and integer programming*. Wiley, 1998.

16. R. Sebastiani. Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224, 2007.

17. T. Terauchi. Dependent types from counterexamples. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 119–130. ACM, 2010.

18. T. Terauchi and H. Unno. Relaxed stratification: A new approach to practical complete predicate refinement. 2015. `http://www.jaist.ac.jp/~terauchi`.

19. G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

20. H. Unno and N. Kobayashi. Dependent type inference with interpolants. In A. Porto and F. J. López-Fraguas, editors, *PPDP*, pages 277–288. ACM, 2009.

21. H. Unno and T. Terauchi. Inferring simple solutions to recursion-free horn clauses via sampling. In *TACAS*, 2015. To appear.

22. H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 75–86. ACM, 2013.