Chronosymbolic Learning: Efficient CHC Solving with Symbolic Reasoning and Inductive Learning

Anonymous author(s)

Under double-blind review

Abstract. Solving Constrained Horn Clauses (CHCs) is a fundamental challenge behind a wide range of verification and analysis tasks. To enhance CHC solving without the laborious task of manual heuristic creation and tuning, data-driven approaches demonstrate significant potential by extracting crucial patterns from a small set of data points. However, at present, symbolic methods generally surpass data-driven solvers in performance. In this work, we develop a simple but effective framework, Chronosymbolic Learning, which unifies symbolic information and numerical data to solve a CHC system efficiently. We also present a simple instance[1] of Chronosymbolic Learning with a data-driven learner and a BMC-styled reasoner[2]. Despite its great simplicity, experimental results show the efficacy and robustness of our tool. It outperforms state-of-the-art CHC solvers on a test suite of 288 arithmetic benchmarks, including some instances with non-linear arithmetic.

1 Introduction

Constrained Horn Clauses (CHCs), a fragment of First Order Logic (FOL), naturally capture the discovery and verification of inductive invariants [1]. CHCs serve as a general format for program safety verification[3]. They are widely used in software verification frameworks including C/C++, Java, Rust, Solidity, and Android verification frameworks [19, 22, 24], modular verification of distributed and parameterized systems [16, 20], type inference [36], and many others [18]. Given the importance of these applications, building an efficient CHC solver holds great significance. Nevertheless, the undecidable nature of CHC solving necessitates tailored adjustments or designs for specific instances, demanding substantial effort and expertise.

Remarkable progress in automating CHC solving has been achieved in recent years. Existing approaches primarily fall into two categories: symbolic-reasoning-based approaches and data-driven induction-based approaches. The former centers around designing novel symbolic reasoning techniques, such as abstraction

[1] The artifact is available on this anonymous link: `https://anonymous.4open.science/r/Chronosymbolic-Learning-343A`

[2] BMC represents for Bounded Model Checking [7].

[3] See Appendix A.2 for details.

refinement [8], interpolation [25], property-directed reachability [3, 11], model-based projections [23], and other techniques. While the latter focuses on reducing CHC solving into a machine learning (ML) problem and then employing proper ML models, such as Boolean functions [29], decision trees (DTs) [15], support vector machines (SVMs) [41], and deep learning models [35]. While data-driven approaches show great promise towards improving CHC solving without the painstaking manual heuristic tuning, data-driven CHC solvers still fall way behind symbolic reasoning-based CHC solvers [37]. Fig. 1 illustrates key differences between these two categories. Symbolic approaches usually maintain two *zones* approximating safe and unsafe "states" of a system represented by CHCs, meticulously updated with soundness guarantees. Data-driven approaches abstract symbolic constraints away by sampling positive and negative data points. The sampling process usually requires some form of *evaluation* of the given constraints, thus, sampled data points are the outcome of complicated interactions among multiple constraints, making data-driven approaches excel at capturing useful global properties. However, as illustrated by the dashed grey line in Fig. 1, the primary drawback of data-driven approaches is that the classifier learned from samples may overlook safe regions that symbolic reasoning can easily identify. Conversely, symbolic approaches are good at precisely analyzing local constraints, delivering superior performance but potentially getting stuck in a local region. Additionally, they struggle to identify essential patterns from data samples, limiting their flexibility in addressing these problems.

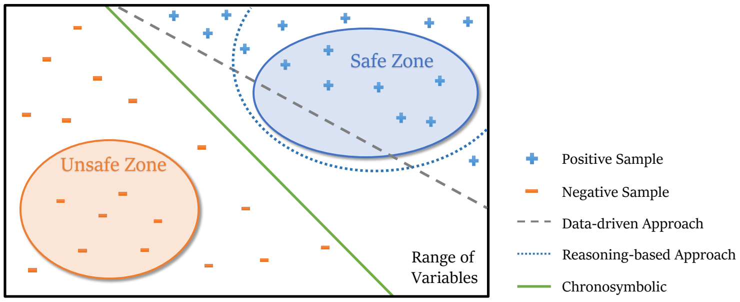

Fig. 1: Overview of different approaches through the lens of learning from positive samples and negative samples.

The motivation for our framework, Chronosymbolic Learning, is to devise a learning framework integrating the strengths of both symbolic and data-driven methods. Similar to data-driven approaches, a *learner* in Chronosymbolic Learning derives classifiers from sampled data points, a process attainable through machine learning approaches. However, rather than utilizing these learned classifiers directly as hypotheses, we augment them with symbolic information summarized by a *reasoner*. Such symbolic information can be viewed as a region of data points *concisely* expressed in a symbolic form, whereas the learned classifiers serve as symbolic representations of *generalized* regions. Instead of combining these two mechanically or parallelly, our method makes the reasoner and learner mutually benefit from each other, as exemplified by the green line of Fig. 1. It also lays the groundwork for seamlessly synergizing data learning power with automated reasoning power, offering a paradigm that guides future endeavors to incorporate cutting-edge ML techniques.

Main Contributions. Our contribution is mainly three-fold. First and foremost, we propose a novel framework, Chronosymbolic Learning. As the name suggests[4], our goal is to solve CHCs with both numerical data samples and symbolic information in a synchronous and synergistic manner. We provide formulations in order to realize this desideratum, establishing the groundwork for the application of advanced techniques in symbolic reasoning and machine learning. Secondly, we build a simple yet potent instance of Chronosymbolic Learning in Python, a standard programming language in the ML community, to substantiate our claims. It comprises a data-driven *learner* interacting with a BMC-styled *reasoner*, alongside a verification oracle as a *teacher*. Lastly, the main experimental results demonstrate the effectiveness and robustness of our tool. It outperforms several state-of-the-art CHC solvers on a test suite of 288 benchmarks, including some instances with non-linear integer arithmetic.

2 Preliminaries

2.1 Constrained Horn Clauses

We discuss standard *First Order Logic* (FOL) formula modulo theory $\mathcal{T}$, with a signature Σ composed by constant symbols $\mathcal{A}$ (e.g., True $\top$ and False $\bot$), function symbols $\mathcal{F}$ (e.g., +, -, `mod`) and predicate symbols $\mathcal{P}$. A *Constrained Horn Clause* (CHC) $\mathcal{C}$ is a FOL formula modulo theory $\mathcal{T}$ in the following form:

$$\forall \mathcal{X} \cdot \phi \wedge p_1(T_1) \wedge \cdots \wedge p_k(T_k) \rightarrow h(T),\ \ k \geq 0, \tag{1}$$

where $\mathcal{X}$ stands for all variables in T_i and T, ϕ represents a fixed constraint over $\mathcal{F}; \mathcal{A}$ and $\mathcal{X}$ w.r.t. some background theory $\mathcal{T}$; p_i, h are predicate symbols[5], and $p_i(T_i) = p_i(t_{i,1}, \cdots, t_{i,n})$ is an application of n-ary predicate symbol p_i with first-order terms $t_{i,j}$ built from $\mathcal{F}, \mathcal{A}$ and $\mathcal{X}$; $h(T)$ could be either $\mathcal{P}$-free (i.e., no predicate symbol appears in h) or akin to the definition of $p_i(T_i)$.

In this work, we explain our framework using the background theory of *Integer Arithmetic* (IA) if it is not specified. For simplification, in our work, the universal quantifier "$\forall$" is omitted when the context is clear; and all terms $t_{i,j}$ represent single variables (e.g., $t_{1,1} = x$, $t_{1,2} = y$).

The structure of a CHC can be viewed as a split based on the implication symbol. The left hand side $\phi \wedge p_1(T_1) \wedge \cdots \wedge p_k(T_k)$ is called *body* and the right hand side $h(T)$ is called *head*. If the body of a CHC contains zero or one predicate symbol, the CHC is called *linear*. Otherwise, it is called *non-linear*.

[4] The name Chronosymbolic Learning is a blend of the terms CHC, number (represented by "no."), symbolic and synchronous.

[5] "predicates" for short

Example 1. A simple CHC system $\mathcal{H}_0$[6] consists of a fact $\mathcal{C}_0$, a non-fact rule $\mathcal{C}_1$ and a query $\mathcal{C}_2$ is as follows:

$$\mathcal{C}_0 : \neg(a \leq 0) \ \wedge \ b \leq a \ \wedge \ c = 0 \ \wedge \ e = 0 \ \wedge \ d = 0 \rightarrow \mathit{inv}(a, b, c, d, e)$$
$$\mathcal{C}_1 : c_1 = 1 + c \wedge d_1 = d + a \wedge e_1 = e + b \wedge \mathit{inv}(a, b, c, d, e) \rightarrow \mathit{inv}(a, b, c_1, d_1, e_1)$$
$$\mathcal{C}_2 : \neg(e \geq a \cdot c) \ \wedge \ \mathit{inv}(a, b, c, d, e) \rightarrow \bot$$

In most cases, the terminology "CHC solving" refers to solving the *satisfiability* (SAT) problem for a *CHC system* that is a set of CHCs containing at least a query and a rule. A *query* $\mathcal{C}_q$ refers to a CHC that has a $\mathcal{P}$-free head, otherwise, it is called a *rule* $\mathcal{C}_r$. Particularly, a *fact* $\mathcal{C}_f$ refers to a rule that has a $\mathcal{P}$-free body. For example, in Example 1, $\mathcal{C}_0$ is a fact and a rule, $\mathcal{C}_1$ is a rule, and $\mathcal{C}_2$ is a query. We define the clauses that both body and head are $\mathcal{P}$-free as *trivial* clauses because they can be simply reduced to $\top$ or $\bot$.

Then, we formally define the satisfiability of a CHC system $\mathcal{H}$:

Definition 1 (Satisfiability of a CHC system). *$\mathcal{H}$ is satisfiable* (SAT) *iff there exists an interpretation of predicates $\mathcal{I}^*$ such that $\forall \mathcal{C} \in \mathcal{H}$, $\mathcal{I}^* [\mathcal{C}]$ is $\top$. Otherwise, we say $\mathcal{H}$ is unsatisfiable* (UNSAT).

Definition 2 (Solution of a CHC system). *A solution of $\mathcal{H}$ consists of its satisfiability and a proof of its satisfiability.*

If $\mathcal{H}$ is SAT, the proof is the corresponding interpretation $\mathcal{I}^*$, and $\mathcal{I}^*[p]$ is called a *solution interpretation* of the predicate p. If $\mathcal{H}$ is UNSAT, the proof is a *refutation* demonstrating the nonexistence of a solution interpretation [2]. The refutation can be seen as a specific value assignment for the variables in $\mathcal{H}$ that makes all the CHCs in $\mathcal{H}$ $\bot$. Therefore, the problem addressed in this work is to *determine a solution for a given CHC system.*

2.2 CHC Solving as a Symbolic Classification Problem

To harness the data samples, we cast the CHC solving problem into a symbolic classification task by utilizing the samples to produce hypotheses. For clarity, we formally define the term *hypothesis*:

Definition 3 (Hypothesis). *A hypothesis of $\mathcal{H}$ is an interpretation $\tilde{\mathcal{I}}$ that is a possible solution interpretation of $\mathcal{H}$.*

To determine a hypothesis for a specific predicate p denoted as $\tilde{\mathcal{I}}[p]$, [41] introduce a lightweight machine learning toolchain that combines DT learning with the results of SVMs. In this data-driven pipeline, positive and negative samples are iteratively classified by SVMs until all samples are correctly categorized. The resulting hyperplanes of SVMs, $f(\boldsymbol{x}) = w^T \boldsymbol{x}_i + b$, serve as *attributes* for the DT. The DT then selects an attribute $f(\boldsymbol{x}_0)$ and a corresponding threshold c in the form of $f(\boldsymbol{x}_0) \leq c$ to create a new node in the way of maximizing the

[6] This is the benchmark `nonlin_mult_2.smt2` in our suite of test benchmarks.

information gain γ. Adjusting the threshold c and pruning the over-complicated attribute combinations enhances generalizability and mitigates overfitting risks. This process forms a DT, effectively segregating all samples into positive and negative categories. The DT generates a learned formula[7], denoted as $\mathcal{L}_p$, which can be arbitrary linear inequalities and their combinations in *disjunctive normal form* (DNF). In our framework, this formula is termed a *partition*: a hypothesis proposed only by the learner.

3 Chronosymbolic Learning

As shown in Fig. 1, pure data-driven and reasoning-based approaches exhibit distinct limitations. The former is completely agnostic to the inherent symbolic information within the CHC system, and relies solely on data samples for hypothesis generation. The latter struggles to *induce* global patterns of solution interpretations, and faces challenges in effectively integrating data samples into its reasoning process. To address the limitations, we propose Chronosymbolic Learning, a modular framework designed to amalgamate the strengths of both approaches and harness the full potential of symbolic information and numerical data concurrently.

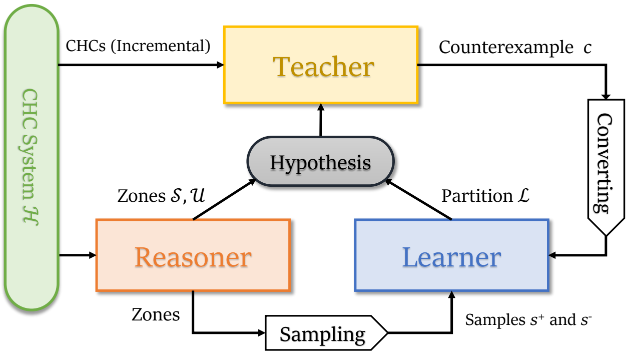

Fig. 2: The architecture of Chronosymbolic Learning.

3.1 Architecture of Chronosymbolic Learning

Chronosymbolic Learning, depicted in Fig. 2, extends the paradigm of "*teacher*[8] *and learner*" [14], which finds a solution to a given CHC system by guessing and checking [13, 33]. In this paradigm, the teacher and learner engage in iterative communication. During each iteration, the learner formulates a *hypothesis* (refer to Definition 3) of an interpretation called *partition*. The

[7] The fundamentals of DT, SVM, and the formula generation is specified in Appendix A.

[8] Also refers to a verification oracle, such as Microsoft Z3.

teacher then verifies the hypothesis and provides instant feedback regarding the hypothesis's correctness. If incorrect, the teacher supplies a *counterexample* (see Definition 8) elucidating the reason for its inaccuracy.

In addition to this paradigm, in Chronosymbolic Learning, the learning procedure is enhanced by a *reasoner*. The reasoner maintains a *safe zone* and an *unsafe zone*, representing symbolic equivalents of *positive and negative samples* respectively. These zones offer three main advantages: 1) They can be integrated into the learner's proposed hypothesis to enhance it; 2) They provide the learner with additional samples; 3) They significantly simplify the UNSAT checking of the CHC system[9]. We will provide an instance of Chronosymbolic Learning, detailing the functionality and intercommunication of its modules as in Fig. 2. The learner we specify in Section 4.1 and reasoner in Section 4.2 can be replaced by any other algorithms capable of generating partitions and zones. See Section 3.4 for the overall algorithm.

3.2 Samples and Zones

To introduce Chronosymbolic Learning, we first conceptualize samples and zones. We shed light on obtaining the samples and zones in Section 4.

Definition 4 (Positive Sample). *A data point s^+ is a positive sample of predicate p in $\mathcal{H}$ iff for each solution interpretation $\mathcal{I}^*$ of $\mathcal{H}$, we have $\mathcal{I}^*[p](s^+) = \top$.*

Definition 5 (Negative Sample). *A data point s^- is a negative sample of p in $\mathcal{H}$ iff for each $\mathcal{I}^*$ of $\mathcal{H}$, we have $\mathcal{I}^*[p](s^-) = \bot$.*

Definition 6 (Implication Sample). *An implication sample $s^\rightarrow$ of body predicates $(p_1, ..., p_n)$ and head predicate h in $\mathcal{H}$ is an $(n+1)$-tuple of data points $(s_1^\rightarrow, \cdots, s_n^\rightarrow, s_h^\rightarrow)$ such that for each solution interpretation $\mathcal{I}^*$ of $\mathcal{H}$, we have $\mathcal{I}^*[p_1](s_1^\rightarrow) \wedge \cdots \wedge \mathcal{I}^*[p_n](s_n^\rightarrow) \rightarrow \mathcal{I}^*[h](s_h^\rightarrow)$.*

Example 2. In $\mathcal{H}_0$ in Example 1, for predicate inv, $s^+ = (1,1,0,0,0)$ is a positive sample, since $s^+[\mathcal{C}_0] : \top \rightarrow inv(1,1,0,0,0)$. $s^- = (2,1,5,0,5)$ is a negative sample, as $s^-[\mathcal{C}_2] : inv(2,1,5,0,5) \rightarrow \bot$. $s^\rightarrow = ((1,1,0,0,0),(1,1,1,1,1))$ is an implication sample, because $s^\rightarrow[\mathcal{C}_1] : inv(1,1,0,0,0) \rightarrow inv(1,1,1,1,1)$.

We now formally define safe and unsafe zones, borrowing the terms "safe" and "unsafe" from program verification.

Definition 7 (Safe and Unsafe Zones). *A safe (unsafe) zone of a predicate p, $\mathcal{S}_p$ ($\mathcal{U}_p$), is a set of positive (negative) samples of p.*

The zones are often symbolically represented as expressions, such as inequalities and equations. They may include zero sample[10], or a finite or infinite number of samples. Notably, a positive sample can also be viewed as a safe zone, while a negative sample can be seen as an unsafe zone.

[9] In program verification parlance, it refers to the unsafe check.

[10] In this case, the zone is $\bot$.

Example 3. In $\mathcal{H}_0$ in Example 1, one safe zone for predicate $inv(v_0, v_1, v_2, v_3, v_4)$ is $\mathcal{S}_{inv} = \neg(v_0 \leq 0) \ \wedge \ v_1 \leq v_0 \ \wedge \ v_2 = 0 \ \wedge \ v_3 = 0 \ \wedge \ v_4 = 0$, since $s^+ \in \mathcal{S}_{inv}$ satisfies $s^+[\mathcal{C}_0] : \top \to inv(s^+)$. One unsafe zone for predicate inv is $\mathcal{U}_{inv} = \neg(v_4 \geq v_0 v_2)$, since $s^- \in \mathcal{U}_{inv}$ satisfies $s^-[\mathcal{C}_2] : inv(s^-) \to \bot$.

3.3 Incorporate Zones within Learning Iterations

Chronosymbolic Learning is a framework that incorporates zones within the learning iterations. A *Chronosymbolic Learner* proposes the hypothesis $\tilde{\mathcal{I}}\,[p_i]$ considering currently reasoned safe zones $\mathcal{S}_{p_i}$, unsafe zones $\mathcal{U}_{p_i}$ from the reasoner with the inductive results (partitions) of the learner $\mathcal{L}_{p_i}$.

Table 1: Several candidates of making hypotheses.

Methods	**Candidate Hypothesis**
BMC-styled	$\tilde{\mathcal{I}}_s\,[p_i] = \mathcal{S}_{p_i} \qquad (2)$
LinearArbitrary-styled	$\tilde{\mathcal{I}}_l\,[p_i] = \mathcal{L}_{p_i} \qquad (3)$
Chronosymbolic without safe zones	$\tilde{\mathcal{I}}_{lu}\,[p_i] = \mathcal{L}_{p_i} \wedge \neg\, \mathcal{U}_{p_i} \qquad (4)$
Chronosymbolic without unsafe zones	$\tilde{\mathcal{I}}_{sl}\,[p_i] = \mathcal{S}_{p_i} \vee \mathcal{L}_{p_i} \qquad (5)$
Chronosymbolic	$\tilde{\mathcal{I}}_{slu}\,[p_i] = \mathcal{S}_{p_i} \vee (\mathcal{L}_{p_i} \wedge \neg\, \mathcal{U}_{p_i}) \quad (6)$

Several promising candidates of the hypothesis $\tilde{\mathcal{I}}\,[p_i]$ that can be made by *Chronosymbolic Learner*[11] are listed in Table 1. Note that Equation 6 simply serves as an example of a Chronosymbolic hypothesis. It can be replaced by any symbolic classification algorithm taking samples and zones as input, providing a symbolic hypothesis as output. We have the following lemma:

Lemma 1. *$\tilde{\mathcal{I}}_{slu} \succeq \{\tilde{\mathcal{I}}_{sl}, \tilde{\mathcal{I}}_{lu}\} \succeq \tilde{\mathcal{I}}_l$, where $A \succeq B$ denotes that A is a solution interpretation of the CHC system $\mathcal{H}$ whenever B is a solution interpretation.*

Lemma 1 shows the order of the feasibility of becoming a solution interpretation. We give a proof sketch for $\tilde{\mathcal{I}}_{sl} \succeq \tilde{\mathcal{I}}_l$ and proofs for others are similar.

Proof. Assume $\tilde{\mathcal{I}}_l\,[p_i] = \mathcal{L}_{p_i}$ is a solution interpretation of $\mathcal{H}$, and then by Definition 4, we have for each positive sample s^+, $\tilde{\mathcal{I}}_l\,[p_i]\,(s^+) = \top$, i.e., $s^+ \in \tilde{\mathcal{I}}_l\,[p_i]$. According to Definition 7, the safe zone $\mathcal{S}_{p_i}$ is a set of positive samples, and we have $\mathcal{S}_{p_i} \subseteq \tilde{\mathcal{I}}_l\,[p_i]$. We now have $\tilde{\mathcal{I}}_{sl}\,[p_i] = \mathcal{S}_{p_i} \wedge \tilde{\mathcal{I}}_l\,[p_i] = \tilde{\mathcal{I}}_l\,[p_i]$ and $\tilde{\mathcal{I}}_{sl}\,[p_i]$ is also a solution interpretation. Thus, we can conclude $\tilde{\mathcal{I}}_{sl} \succeq \tilde{\mathcal{I}}_l$. □

In practice, Equation 6 generally delivers the optimal result among the candidates, as it encapsulates more information summarized by both the learner and the reasoner[12]. Nevertheless, this doesn't hold true for every individual instance,

[11] This is the procedure `makeHypothesis()` in Algorithm 1.

[12] Appendix B provides additional theoretical analysis on why Chronosymbolic Learning performs better from the perspective of the sample and solution space.

since the zones may carry "distractive" information[13]. To address this issue, we can manually apply adequate strategies on each instance, or use a *scheduler* to alternate those candidate hypotheses over time. Our experiments will demonstrate the performance improvement achieved by employing various strategies.

3.4 Overall Algorithm

Algorithm 1 Chronosymbolic Learning ($\mathcal{H}$)

```
input: A CHC system $\mathcal{H} = \{\mathcal{C}_0, \cdots, \mathcal{C}_n\}$
 1: • Initialize safe and unsafe zones       $\forall\ p_i,\ (\mathcal{S}_{p_i}, \mathcal{U}_{p_i}) \leftarrow (\bot, \bot)$
 2: • Initialize the hypotheses              $\forall\ p_i,\ \tilde{\mathcal{I}}_{p_i} \leftarrow \top$
 3: • Initialize UNSAT flag                  $is\_unsat \leftarrow \bot$
 4: • Initialize the dataset                 $\mathcal{D} \leftarrow \varnothing$
 5: while not $is\_unsat$ and $\exists\ \mathcal{C}_i$, not SMTCheck($\tilde{\mathcal{I}}\,[\mathcal{C}_i]$) do
 6:    ○ Reasoning and UNSAT checking        $(\mathcal{S}, \mathcal{U}, is\_unsat, \tilde{\mathcal{I}}) \leftarrow$ reason($\mathcal{H}, \mathcal{S}, \mathcal{U}$)
 7:    ○ Sample from zones and add to $\mathcal{D}$     $\mathcal{D} \leftarrow$ sampling($\mathcal{S}, \mathcal{U}, \mathcal{D}$)
 8:    for each $\mathcal{C}_i \in \mathcal{H}$ do
 9:       while not SMTCheck($\tilde{\mathcal{I}}\,[\mathcal{C}_i]$) do
10:          • Find counterexample(s)        $c \leftarrow$ SMTModel($\tilde{\mathcal{I}}\,[\mathcal{C}_i]$)
11:          • Convert counterexample(s) to samples     $s \leftarrow$ converting(c)
12:          • UNSAT checking                $is\_unsat,\ \tilde{\mathcal{I}} \leftarrow$ checkUNSAT($\mathcal{D}$, s)
13:          • Add samples to $\mathcal{D}$              $\mathcal{D} \leftarrow s$, if not $is\_unsat$
14:          • Learn a partition             $\mathcal{L} \leftarrow$ learn($\mathcal{D}$)
15:          • Update the hypothesis         $\tilde{\mathcal{I}} \leftarrow$ makeHypothesis($\mathcal{S}, \mathcal{U}, \mathcal{L}$)
16:       end while
17:    end for
18: end while
output: A solution ($is\_unsat$, $\tilde{\mathcal{I}}$) for the CHC system $\mathcal{H}$
```

The overall algorithm is outlined in Algorithm 1, where the solid bullet points denote "mandatory" and the unfilled bullet points indicate "optional", suggesting that they need not be executed in every iteration.

The algorithm takes a CHC system $\mathcal{H}$ as input and outputs a solution for it. Initialization of zones, hypotheses, UNSAT flag, and the dataset occurs in lines 1-4. The outer while loop, spanning lines 5 to 18, checks if a solution of $\mathcal{H}$ has been found. If not, it initiates a new epoch and continues solving. In line 5, `SMTCheck(`$\tilde{\mathcal{I}}\,[\mathcal{C}_i]$`)` calls the backend SMT solver to check the satisfiability of a CHC $\mathcal{C}_i$ under current hypothesis interpretation $\tilde{\mathcal{I}}$. In line 6, the reasoner refines the zones and checks if the zones overlap (see Section 4.2). Line 7 involves the *sampling* procedure described in Section 4.1.

[13] Such information may change the solution space of the backend SMT solver, and the teacher may also end up returning counterexamples that lead to less progress for the learner.

The for loop from lines 8 to 17 iteratively finds an UNSAT CHC and refines the hypothesis until it becomes SAT[14]. In the inner while loop in lines 10-13, we find one or a batch of counterexamples, convert them into samples, and add them to the dataset after the UNSAT checking[15] passes. `SMTModel(`$\tilde{\mathcal{I}}\,[\mathcal{C}_i]$`)` in line 10 calls the backend SMT solver to return a counterexample that elucidates why the current hypothesis is invalid. Following dataset updates, in lines 14-15, the learner induces a new partition. We then update the Chronosymbolic hypothesis by combining the information of the zones and the partition, as described in Section 3.3. The new hypothesis is guaranteed to be better off in terms of the partial order in Lemma 1 because it incorporates the fresh information collected in this iteration.

4 Design of Learner and Reasoner

In this section, we provide a simple instance of Chronosymbolic Learning, utilizing standard procedures as illustrative examples to demonstrate how they can be integrated into our framework.

4.1 Learner: Data-Driven CHC Solving

The *learner* module in our framework leverages an induction-based and CEGAR-inspired [8] CHC solving scheme. It has two sub-modules: the *dataset* and the *machine learning toolchain.* The dataset stores the positive and negative samples and the corresponding predicates for inductive learning. The machine learning toolchain (an example introduced in Section 2.2) takes the samples in the dataset as input and outputs a *partition* that can correctly classify all these samples.

Converting: From Counterexamples to Samples. The positive and negative samples in our framework are *converted* from *counterexamples.* As in Fig. 2, the counterexample provided by the teacher is essential in the learning loop, since it offers the learner information about why the current hypothesis is incorrect and how to improve it. Here we formally define the term "counterexample".

Definition 8 (Counterexample). *A counterexample* $c = ((s_{p_1}, \cdots, s_{p_k}), s_h)$ *for a CHC* $\mathcal{C}$ *and an interpretation* $\mathcal{I}$ *is a set of data points*[16] *such that under* c*,* $\mathcal{I}\,[\mathcal{C}'] = \bot$*, where* $\mathcal{C}'$ *is the same constraint as* $\mathcal{C}$ *but without the quantifier.*

Example 4. In $\mathcal{H}_0$ in Example 1, given that the current hypothesis of predicate $inv(v_0, v_1, v_2, v_3, v_4)$ is $\tilde{\mathcal{I}}\,[inv] = v_3 \leq 1$, a counterexample for $\mathcal{C}_1$ could be $c = (((2, 0, 0, 0, 0)), (2, 0, 1, 2, 0))$, as it makes $\tilde{\mathcal{I}}[\mathcal{C}_1] : \top \to \bot$.

[14] An alternative design choice is to update the hypothesis only once and move to the next UNSAT CHC, which achieves better results in some cases.

[15] For simplicity, the refutation proof generated when checking UNSAT is also represented as $\tilde{\mathcal{I}}$ in Algorithm 1.

[16] They can also be extended to zonal representation.

Counterexamples can be converted into positive, negative, or implication samples through the *converting* procedure in Fig. 2 and Algorithm 1. The following converting scheme maps counterexamples into positive, negative, and implication samples based on the CHC type. Essentially, this approach samples the CHC system in three directions: forward, backward, and midway.

Lemma 2 (Sample Converting). *A fact's counterexample can provide a positive sample for the head predicate. A linear query's counterexample can provide a negative sample for the body predicate. A non-fact rule's counterexample can provide an implication sample of the body predicate(s) and the head predicate.*

To handle implication samples, we introduce "tentative sample[17]":

Definition 9 (Tentative Sample). *A tentative positive (negative) sample $\tilde{s}^+$ ($\tilde{s}^-$) of predicate p is a data point such that, the learner tentatively deems it to be a positive (negative) sample, but it may not satisfy Definition 4 (5).*

Lemma 3 (Implication Sample Converting). *For an implication sample $s^{\rightarrow}$, if the n data points in $(s_1^{\rightarrow}, \cdots, s_n^{\rightarrow})$ are all positive samples (or in a safe zone), then $s_h^{\rightarrow}$ is a positive sample. If the data point $s_h^{\rightarrow}$ is a negative sample (or in an unsafe zone), and $n = 1$, the data point $s_1^{\rightarrow}$ is a negative sample. Otherwise, the sample can only be converted into tentative positive or negative samples, whose tentative categories make the logical implication hold.*

Empirically, converting all tentative samples into negative tentative samples is a decent design choice. As these samples are not guaranteed to be positive or negative, periodic clearing is necessary to avoid keeping the "wrong guess" in a specific tentative category forever. Following this conversion, only positive and negative samples remain visible to the learner, enabling natural adaptation with off-the-shelf machine learning tools for binary classification like DT and SVM.

Sampling: Obtain Data Points from Zones. To find extra data points and maximize the usage of the zones, we provide an alternative strategy of data collection, i.e., *sampling* data points from zones. This procedure involves selecting a sample from a given zone. Typically, it calls the backend SMT solver with a safe (unsafe) zone as the input constraint. The SMT solver returns positive (negative) data points within the zone while ensuring the avoidance of duplicates already present in the dataset.

UNSAT Checking. Two lemmas show when the learner determines a CHC system as UNSAT.

Lemma 4 (Sample-Sample Conflict). *If there exists a sample s for p in $\mathcal{H}$ that is both a positive and a negative sample simultaneously, then $\mathcal{H}$ is UNSAT.*

[17] It is extensible to *tentative zones* to make an IC3-styled reasoner possible.

Proof. According to the Definition 4 and 5, if s is both a positive and negative sample, to make $\mathcal{H}$ SAT, we have $p(s) = \top$ and $p(s) = \bot$, which contradict each other. Hence, $\mathcal{H}$ is UNSAT. □

Lemma 5 (Sample-Zone Conflict). *If there exists a sample s for p in $\mathcal{H}$ that is a positive sample and is in an unsafe zone, or is a negative sample and is in a safe zone, then $\mathcal{H}$ is UNSAT.*

Proof. With Definition 7, sample-zone conflict can be reduced to sample-sample conflict, the case in Lemma 4. □

4.2 Reasoner: Zones Discovery

For the *reasoner*, we adopt a simple BMC-styled[18] logical reasoning procedure for inferring safe and unsafe zones in our instance of Chronosymbolic Learning. The role of the reasoner is not to deduce a complete hypothesis but to furnish a "partial solution" in the form of safe and unsafe zones, collaborating well with the learner. The zones condense information within a bounded number of transitions. In our tool, the reasoner initializes the zones, expands the zones through image (or pre-image) computation, and does UNSAT checking.

In each of the subsequent lemmas, all variables utilized in the discussed CHC are denoted as $\mathcal{X}$. Among them, variables used by the discussed predicate are denoted as $\mathcal{X}_\mu \subseteq \mathcal{X}$. The other variables are called "*free variables*", denoted as $\mathcal{X}_\varphi = \mathcal{X} \setminus \mathcal{X}_\mu \subseteq \mathcal{X}$.

Initialization of Zones. We provide several lemmas about initiating zones.

Lemma 6 (Initial Safe Zones). *A fact $\mathcal{C}_f : \phi_f \to h_f(T)$ produces a safe zone for h_f: $\mathcal{S}^0_{h_f}(T) = \exists \mathcal{X}_\varphi, \phi_f$.*

Lemma 7 (Initial Unsafe Zones). *A linear query $\mathcal{C}_q : \phi_q \wedge p_q(T) \to \bot$ produces an unsafe zone for p_q: $\mathcal{U}^0_{p_q}(T) = \exists \mathcal{X}_\varphi, \phi_q$.*

Expansion of the Zones. Zones can be expanded in the following ways:

Lemma 8 (Forward Expansion). *From given safe zones $\mathcal{S}^m_{p_i}$, we can expand them in one forward transition by a non-fact rule $\mathcal{C}_r : \phi \wedge p_1(T_1) \wedge \cdots \wedge p_k(T_k) \to h(T)$ to get an expanded safe zone $\mathcal{S}^{m+1}_h$, where $\mathcal{S}^{m+1}_h(T) = \exists \mathcal{X}_\varphi, \phi \wedge \mathcal{S}^m_{p_1}(T_1) \wedge \cdots \wedge S^m_{p_k}(T_k)$.*

Lemma 9 (Backward Expansion). *From a given unsafe zone $\mathcal{U}^m_h$, we can expand it in one backward transition by a non-fact linear rule $\mathcal{C}_r : \phi \wedge p(T_0) \to h(T)$ to get an expanded unsafe zone $\mathcal{U}^{m+1}_p$, where $\mathcal{U}^{m+1}_p(T_0) = \exists \mathcal{X}_\varphi, \phi \wedge \mathcal{U}^m_h$.*

In the context of program verification, the forward expansion incorporates more information about the reached states. The backward expansion expands the set of states that the program is deemed unsafe when reached. Note that the expansion operation can be computationally expensive, and excessive expansion leads to a heavy burden for the backend SMT solver.

[18] The reasoner may resemble other symbolic CHC solvers like IC3.

UNSAT Checking. We further discuss when the reasoner determines UNSAT.

Lemma 10 (Zone-Zone Conflict). *If there exists a predicate p in $\mathcal{H}$ that its safe zone $\mathcal{S}_p$ and unsafe zone $\mathcal{U}_p$ overlap, i.e., $\mathcal{S}_p \wedge \mathcal{U}_p$ is SAT, then $\mathcal{H}$ is UNSAT.*

Proof. According to Definition 7, we sample one data point from the overlap of zones, and we can then apply Lemma 5 to prove Lemma 10. □

5 Experiments

5.1 Experimental Settings

Benchmarks. Our benchmarking dataset comprises a collection of widely-used arithmetic CHC benchmarks from related works [10, 12, 15] and SV-COMP[19], collected by FreqHorn [12]. It is a suite of 288 instances in total, consisting of 235 safe ones and 53 unsafe ones. The instances are lightweight arithmetic CHC systems, yet many demand non-trivial effort to determine the solution interpretation due to the arithmetic complexity. Most instances are under linear arithmetic, while a few necessitate handling non-linear arithmetic. We also discuss results on CHC-COMP-22[20], LIA track. See Appendix D.1 for details.

Hyperparameters. In SVM, the C parameter is set to 1. The upper limit of the coefficients in the hyperplanes is set to 5. For DT, we use C5.0 as a default option. For the dataset, the best result we achieved is to use option A in Appendix C.1, where $a = 50$, $b = 50$. For reasoner, the l_1, l_2 and l_3 in Appendix C.2 are 700, 500 and 1500[21] respectively.

Experiment Setup. We imposed a timeout of solving a CHC system instance as 360 seconds across all tools. To account for the inherent randomness in the tools, we repeat three times for each experiment and report the best result[22].

5.2 Baselines

State-of-the-art CHC solvers can be classified into three categories: reasoning-based, synthesis-based, and induction-based solvers (see Section 6). We compare our method against the representatives of the three types of solvers as follows:

LinearArbitrary. LinearArbitrary [41] is the state-of-the-art induction-based solver that learns inductive invariants from counterexample-derived samples using SVM and DT. We use the default hyperparameters in the paper and code[23].

[19] `http://sv-comp.sosy-lab.org/`

[20] https://chc-comp.github.io/

[21] More information on the configuration, like the scheduler mentioned in Section 3.3, is shown at `https://anonymous.4open.science/r/Chronosymbolic-Learning-343A/experiment/result_safe_summary.log`.

[22] We include results showcasing how random seeds affect the performance in Appendix D.3.

[23] `https://github.com/GaloisInc/LinearArbitrary-SeaHorn`

FreqHorn. FreqHorn[24] [12] is a representative of synthesis-based solvers, synthesizing invariants using frequency distribution of patterns. It cannot determine the CHC system as UNSAT, and it resorts to a separate procedure "Freqhorn-expl" to do that. In the experiment, we adhere to the default settings in [12].

Spacer and GSpacer. Spacer [23] is a powerful SMT-based CHC-solving engine serving as the default CHC engine in Microsoft Z3[25] [9]. The experiment is conducted under the default settings of Microsoft Z3 4.8.11.0. GSpacer[26] [37] achieves state-of-the-art on many existing CHC benchmarks by extending with some well-designed global guidance heuristics. To achieve the best result, we enable all the heuristics, including the subsume, concretize, and conjecture rules.

5.3 Performance Evaluation

The performance comparison [27] is shown in Table 2. "Chronosymbolic-single" configuration stands for the best result from a single run of our solver, where all instances are tested using a fixed set of hyperparameters. "Chronosymbolic-cover" encompasses all solved benchmarks achieved through 13 runs[28] using different strategies and hyperparameters, demonstrating the potential capability of our framework when selecting an adequate configuration for each instance. Timing information for this setting is not included because the time to solve a certain benchmark varies among different runs. Fig. 3 shows the solved instances for compared methods as the running time increases, and Fig. 4 shows the runtime comparison per instance.

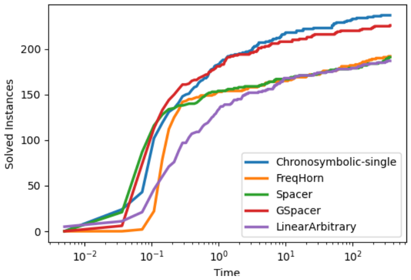

Fig. 3: The solved instances for different methods as the running time increases.

Our tool, even in the Chronosymbolic-single setting, solves 17 more instances than the best competing solver, GSpacer, and generally outperforms other solvers in terms of speed, even though it is implemented in Python. This result shows that our tool performs considerably even without careful tuning for each CHC instance. In the Chronosymbolic-cover setting, our approach shows significant improvements, emphasizing the need for tailored strategies for different instances. Among other approaches, GSpacer performs the best on solved instances and time consumed, as reported in prior literature. Nevertheless, Spacer, GSpacer,

[24] `https://github.com/freqhorn/freqhorn`

[25] `https://github.com/Z3Prover/z3`

[26] `https://github.com/hgvk94/z3/commits/ggbranch`

[27] For detailed timing of each instance, see `https://anonymous.4open.science/r/Chronosymbolic-Learning-343A/experiment/comparison.xlsx`

[28] The details are included in Appendix D.2.

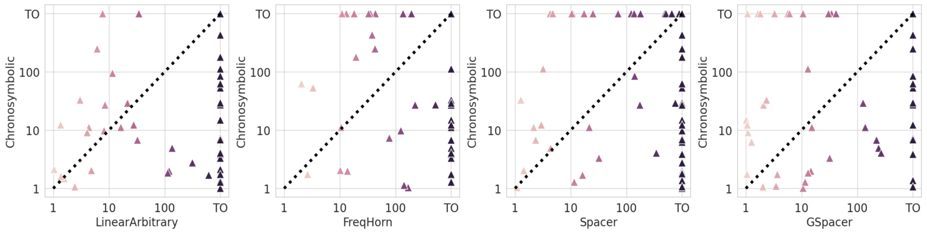

Fig. 4: Running time comparison of Chronosymbolic Learning and baselines, where the timeout (TO) is 360 seconds. The points below the diagonal indicate instances on which Chronosymbolic-single outperforms baselines. Here, we only plot non-trivial instances (i.e., solving time takes ≥ 1 second for both solvers).

Table 2: Performance evaluation. “#total” stands for the total number of solved instances and “percentage” for the percentage of solved ones among all 288 instances, “#safe” and “#unsafe” for solved ones among 235 safe instances and 53 unsafe instances respectively. For timing, “avg-time” is the average time consumed on each instance (including timeout or crashed ones), and “avg-time-solved” is the average time consumed on each solved instance.

Method	#total	percentage	#safe	#unsafe	avg-time (s)	avg-time-solved (s)
LinearArbitrary	187	64.93%	148	39	135.0	13.48
FreqHorn	191	66.32%	191	0	129.1	11.80
FreqHorn-expl	50	17.36%	0	50	299.5	13.57
Spacer	184	63.89%	132	52	132.8	15.30
GSpacer	220	76.39%	174	46	83.50	7.83
Chronosymbolic-single	**237**	**82.29%**	**189**	**48**	**68.33**	**7.51**
Chronosymbolic-cover	**252**	**87.50%**	**204**	**48**	**-**	**-**

and LinearArbitrary struggle with most instances involving non-linear arithmetic. FreqHorn can solve some of them, but fails in all unsafe instances without the resort to another procedure “expl”. Our approach can also handle some simple non-linearity because our reasoner can provide zones with non-linearity.

From Fig. 3, we note that Chronosymbolic-single outperforms FreqHorn and LinearArbitrary under any timeout within 360 seconds, showing the enhanced efficiency of our method. It is also expected that Spacer and GSpacer perform the best under an extremely short time limit as they do not involve time-consuming inductive learning.

Table 3: Evaluation on CHC-COMP-2022-LIA. “#total” and “percentage” stand for the total number and the percentage of solved instances among 499 instances respectively.

Method	#total	percentage
LinearArbitrary	156	31.26%
FreqHorn	123	24.65%
Spacer	261	52.30%
GSpacer	318	63.73%
Chronosymbolic-single	197	39.48%

For CHC-COMP, most benchmarks represent large transition systems with a substantial number of Boolean variables. Reasoning-based approaches inherently suit these benchmarks better, and unsurprisingly, invoking the machine learning toolchain in each iteration is an inefficient design choice. An experiment shows that by filtering out benchmarks in CHC-COMP-22-LIA with excessively large sizes (e.g., exceeding 100 rules or 200 variables), our results are on par with the state-of-the-art (Chronosymbolic-cover 129/208 vs. GSpacer 130/208).

5.4 Ablation Study

We further study the effectiveness of the critical parts of our framework: the learner and the reasoner. For the reasoner, we compare the result when safe zones are not provided, when unsafe zones are not provided, and when neither are provided. Additionally, we assess the performance when entirely removing the learner and running our tool solely with the reasoner. Throughout this ablation study, all other settings remain consistent with Chronosymbolic-single.

Table 4: Different configurations of Chronosymbolic Learning.

Configuration	#total	percentage	#safe	#unsafe	avg-time (s)	avg-time-solved (s)
without safe zones	228	79.17%	183	45	78.56	9.11
without unsafe zones	218	75.69%	173	45	94.75	12.87
without both zones	211	73.26%	166	45	93.84	10.09
without learner	131	45.49%	96	35	196.3	0.16
parallel	216	75.00%	180	36	-	-
Chronosymbolic-single	**237**	**82.29%**	**189**	**48**	**68.33**	**7.51**

From Table 4, as expected, Chronosymbolic-single comprehensively outperforms all other configurations, which shows that each component of our framework is indispensable. It is also clear that each zone provides useful information to the hypothesis, making the result of Chronosymbolic without both zones worse than without safe and unsafe zones. As discussed in Section 4.2 and Appendix , our lightweight reasoner is intended to aid the learner. Thus, the result of not utilizing a learner yields the poorest result, but at a faster computation speed. The result also shows that our learner[29] performs much better than LinearArbitrary, as we have a more comprehensive converting and sampling scheme.

Another experiment in Table 4, "parallel", reveals that if we run our learner and reasoner *individually and simultaneously* for 360 seconds, they only solve a total of 180 safe and 36 unsafe instances. This number is significantly lower than what our proposed method achieves, which underscores the mutual benefit of the reasoner and learner in Chronosymbolic Learning.

6 Related Work

CHC solving has garnered extensive research, offering numerous artifacts for applications like software model checking, verification, and safe inductive invariant generation. Modern CHC solvers primarily rely on three categories of techniques[30]. 1) *Symbolic Reasoning.* Solvers in this category [3, 7, 21, 23, 25,

[29] "Without both zones" indicates that the reasoner does not participate.

[30] See Appendix A.1 for details.

37, 38] maximize the power of logical reasoning and utilize a series of heuristics to accelerate the reasoning engine. 2) *Synthesis.* Methods in this category [10, 12, 28, 31, 35, 40] typically reduce the CHC solving problem into an invariant synthesis problem, which aims to construct an inductive invariant under the *semantic* and *syntactic* constraints. 3) *Induction.* Instead of explicitly constructing the interpretation of unknown predicates, induction-based CHC solvers [5, 8, 14, 15, 27, 32, 33, 39, 41] directly learn such an interpretation by induction learning from data.

7 Future Work

Despite the success, there are several avenues for future work. Firstly, our learner primarily generates linear expressions (and some simple non-linear operators like `mod`), necessitating adaptation for broader abstract domains. It is also crucial to develop and integrate machine learning algorithms that are specifically tailored for this problem within the framework (e.g., how we can obtain a good partition or classifier with the existence of samples and zones). Second, our strategy for identifying safe and unsafe zones is primitive, prompting the exploration of advanced logical reasoning algorithms like Spacer-style approaches, where zone generalization and summarization should be carefully balanced. Thirdly, we acknowledge the influence of different learning strategies on benchmark solving (as Chronosymbolic-cover outperforms Chronosymbolic-single by a large margin) and intend to conduct comprehensive analyses for automated strategy selection. Fourthly, a comprehensive evaluation metric that considers both the number of solved instances and their efficiency is essential for a thorough assessment of solver performance in CHC solving. Lastly, currently, our tool only uses elementary algorithms for zone reasoning, which is not optimized for large CHC systems (e.g., many instances in CHC-COMP). We plan to dedicate efforts to engineering improvements and explore ways to enhance efficiency in downstream machine learning procedures, particularly when handling numerous Boolean variables.

8 Conclusion

In this work, we propose Chronosymbolic Learning, a framework that can synergize reasoning-based techniques with data-driven CHC solving in a reciprocal manner. We also provide a simple yet effective instance of Chronosymbolic Learning to demonstrate its functionality and showcase its potential. Our experiments, conducted on 288 commonly used arithmetic CHC benchmarks, reveal that our tool outperforms several state-of-the-art CHC solvers, encompassing reasoning-based, synthesis-based, and induction-based methods.

References

1. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II, pp. 24–51. Springer (2015)
2. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: International Static Analysis Symposium. pp. 105–125. Springer (2013)
3. Bradley, A.R.: Sat-based model checking without unrolling. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer (2011)
4. Breiman, L., Friedman, J., Olshen, R., Stone, C.: Classification and regression trees. statistics/probability series (1984)
5. Champion, A., Kobayashi, N., Sato, R.: Hoice: An ice-based non-linear horn clause solver. In: Asian Symposium on Programming Languages and Systems. pp. 146–156. Springer (2018)
6. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology **2**, 27:1–27:27 (2011), software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
7. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal methods in system design **19**(1), 7–34 (2001)
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification. pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
9. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
10. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. Acm Sigplan Notices **48**(10), 443–456 (2013)
11. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011. pp. 125–134. FMCAD Inc. (2011)
12. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained horn clauses using syntax and data. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9. IEEE (2018)
13. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/java. In: FME 2001: Formal Methods for Increasing Software Productivity: International Symposium of Formal Methods Europe Berlin, Germany, March 12–16, 2001 Proceedings. pp. 500–517. Springer (2001)
14. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Ice: A robust framework for learning invariants. In: International Conference on Computer Aided Verification. pp. 69–87. Springer (2014)
15. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. ACM Sigplan Notices **51**(1), 499–512 (2016)
16. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 405–416. ACM (2012)

17. Gu, Y., Tsukada, T., Unno, H.: Optimal chc solving via termination proofs. Proceedings of the ACM on Programming Languages **7**(POPL), 604–631 (2023)
18. Gurfinkel, A.: Program verification with constrained horn clauses. In: Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I. pp. 19–29. Springer (2022)
19. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 343–361. Springer (2015)
20. Gurfinkel, A., Shoham, S., Meshman, Y.: SMT-based verification of parameterized systems. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 338–348. ACM (2016)
21. Hojjat, H., Rümmer, P.: The eldarica horn solver. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–7. IEEE (2018)
22. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: A framework for verifying java programs. In: Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28. pp. 352–358. Springer (2016)
23. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: International Conference on Computer Aided Verification. pp. 846–862. Springer (2013)
24. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for rust programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021)
25. McMillan, K.L.: Interpolation and sat-based model checking. In: International Conference on Computer Aided Verification. pp. 1–13. Springer (2003)
26. Miné, A.: The octagon abstract domain. Higher-order and symbolic computation **19**(1), 31–100 (2006)
27. Nguyen, T., Antonopoulos, T., Ruef, A., Hicks, M.: Counterexample-guided approach to finding numerical invariants. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 605–615 (2017)
28. Padhi, S., Sharma, R., Millstein, T.: Data-driven precondition inference with learned features. ACM SIGPLAN Notices **51**(6), 42–56 (2016)
29. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 42–56. ACM (2016)
30. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
31. Riley, D., Fedyukovich, G.: Multi-phase invariant synthesis. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 607–619 (2022)
32. Salzberg, S.L.: C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993 (1994)
33. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22. pp. 574–592. Springer (2013)

34. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. Advances in Neural Information Processing Systems **31** (2018)
35. Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2Inv: A deep learning framework for program verification. In: International Conference on Computer Aided Verification. pp. 151–164. Springer (2020)
36. Tan, B., Mariano, B., Lahiri, S.K., Dillig, I., Feng, Y.: SolType: refinement types for arithmetic overflow in solidity. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022)
37. Vediramana Krishnan, H.G., Chen, Y., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: International Conference on Computer Aided Verification. pp. 101–125. Springer (2020)
38. Vizel, Y., Gurfinkel, A.: Interpolating property directed reachability. In: International Conference on Computer Aided Verification. pp. 260–276. Springer (2014)
39. Xu, R., He, F., Wang, B.Y.: Interval counterexamples for loop invariant learning. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 111–122 (2020)
40. Yao, J., Ryan, G., Wong, J., Jana, S., Gu, R.: Learning nonlinear loop invariants with gated continuous logic networks. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 106–120 (2020)
41. Zhu, H., Magill, S., Jagannathan, S.: A data-driven chc solver. ACM SIGPLAN Notices **53**(4), 707–721 (2018)

Appendix

A Extended Preliminary

A.1 Extended Related Work

Modern CHC solvers primarily rely on the following three categories of techniques.

Symbolic Reasoning. Solvers in this category maximize the power of logical reasoning and utilize a series of heuristics to accelerate the reasoning engine. The classic methods are mostly SAT-based. *Bounded Model Checking* (BMC) [7] encodes the initial states, transitions, and bad states into logical variables, and unfolds k transitions to form a logical formula. However, it only provides a *bounded* proof that can only show the correctness within a certain number of transitions. *Craig Interpolation* (CI) [25] tries to find an "interpolant" that serves as an over-approximating summary of reachable states. If such an interpolant is a fixed-point, then an *unbounded* proof (inductive invariant) is found. *ELDARICA* [21] is an instance of maximizing the power of CI, as it constructs an abstract reachability graph that would witness the satisfiability of CHCs through interpolation. IC3 [3] and PDR [38] greatly improve the performance of finding the unbounded proof by *incrementally* organized SAT-solving. It keeps an over-approximation of unsafe states by recursively blocking them and generalizing the blocking lemmas. Modern solvers embrace the power of SMT-solving to make them extensible to a broader context. *Spacer* [23] is a representative work combining SMT-solving techniques with IC3/PDR-styled unbounded model checking, which is also the current default CHC solver in Microsoft Z3 [9]. It keeps an under-approximation of currently reached states and an over-approximation of unsafe states. *GSpacer* [37] adds global guidance heuristics to enhance Spacer.

Synthesis. Methods in this category typically reduce the CHC solving problem into an invariant synthesis problem, which aims to construct an inductive invariant under the *semantic* and *syntactic* constraints. Many of such methods are heavily inspired by the concept of *Counterexample Guided Inductive Synthesis* (CEGIS), where the synthesizer is forced to produce proposals consistent with the counterexamples collected so far. *Code2Inv* [34, 35] uses a graph neural network to encode the program snippet, and applies reinforcement learning to construct the inductive invariant. *FreqHorn* [12] first generates a grammar that is coherent to the program and a distribution over symbols, and then synthesizes the invariant using the distribution. *InvGen* [10] combines data-driven Boolean learning and synthesis-based feature learning. *G-CLN* [40] uses a template-based differentiable method to generate loop invariants from program traces. It can handle some non-linear benchmarks, but manual hyperparameter tuning for each instance is required, which hampers its generality. In this category, solvers struggle to scale up to CHCs where multiple invariants exist.

Induction. Instead of explicitly constructing the interpretation of unknown predicates, induction-based CHC solvers directly learn such an interpretation by induction. The *ICE-learning* framework [14] provides a convergent and robust learning paradigm. HoIce [5] extends the ICE framework to non-linear CHCs. *ICE-DT* [15] introduces a DT algorithm that adapts to a dataset with positive, negative, and implication samples. *Interval ICE-DT* [39] generalizes samples generated from counterexamples based on the UNSAT core technique to closed intervals, which enhances their expressiveness. *LinearArbitrary* [41] is the most relevant work to ours. It collects samples by checking and unwinding CHCs, and learns inductive invariants from them using SVM and DT [32] within the CEGAR-like [8] learning iterations. But this approach, due to the "black-box" nature, is completely agnostic to the CHC system itself, and much immediate prior knowledge needs to be relearned by the machine learning toolchain.

A.2 CHCs and Program Safety Verification

An essential application of CHCs is to act as a general format for program safety verification. This generality extends to any imperative programming language and any safety constraints specified in First-Order Logic (FOL). To validate a program, we often use a proof system that generates logical formulas called verification conditions (VCs). Validating VCs implies the correctness of the program. It is also common for VCs to include auxiliary predicates, such as *inductive invariants*. Hence, it is natural that the VCs can be in CHC format, and checking the correctness of the program can be converted to checking the satisfiability of the CHCs.

The conversion process from program verification to CHC solving is outlined as follows. Consider the program to be verified as a *Control-Flow Graph* (CFG) [1]. A *Basic Blocks* (BBs) in the program is a vertex of the CFG, equivalent to a predicate in the CHC system. Such predicates can be seen as the summary of the BBs, which corresponds to "*inductive invariants*" in program verification parlance. The edges of the CFG describe the transitions between the BBs. CHCs can encode those transitional relations, in the direction that starts from the BBs of body predicates to the BB of the head predicate. Then, the solution of the result CHC system corresponds to the correctness of the program. Such conversion can be fully automated, and numerous tools are developed for different programming languages, such as *SeaHorn* [19] and *JayHorn* [22].

Table 5 lists a mapping of program verification concepts to CHC solving concepts.

A.3 Support Vector Machine

Support Vector Machines (SVMs) are widely used supervised learning models, especially in small-scale classification problems. In typical SVMs, the quality of a candidate classifier (also called a *hyperplane*) is measured by the margin, defined as its distance from the closest data points, which is often called the support

Table 5: Mapping of program verification concepts to CHC solving concepts.

Program Verification Concept	**CHC Solving Concept**
Verification Conditions (VCs)	CHC system $\mathcal{H}$
Initial program state	Fact $\mathcal{C}_f$
Transition	Rule $\mathcal{C}_r$
Assertion Unsafe condition	Query $\mathcal{C}_q$
Inductive invariant Loop invariant	Solution Interpretation $\mathcal{I}^*[p]$ to predicate p
Counterexample trace	Refutation proof

vectors in a vector space. In our work, we only discuss the case that the data are linearly separable, and linear SVMs can be adopted.

In practice, to prevent over-fitting, it is effective to add some slack variables to make the margin "soft". The optimizing target of such "soft-margin" linear SVM is as follows:

$$\begin{gathered}\min_{w,\xi_i} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \xi_i, \\ \text{subject to } y_i \left(w^T \boldsymbol{x}_i + b \right) \geq 1 - \xi_i, \xi_i \geq 0, \forall i = 1, \ldots, n,\end{gathered} \tag{7}$$

where $w^T \boldsymbol{x}_i + b = 0$ is the hyperplane, ξ_i is the i-th slack variable, C is a hyperparameter that controls the penalty of the error terms. Larger C leads to a larger penalty for wrongly classified samples.

A.4 Decision Tree

Decision Tree (DT) is a classic machine learning tool. It provides an explicit, interpretable procedure for classification problems.

The DT algorithms in our tool make a decision based on the *information gain*. We denote the positive and negative samples as S^+ and S^-, and all samples as $S = S^+ \cup S^-$. Then we define the *Shannon Entropy* $\epsilon(S)$:

$$\epsilon(S) = -\frac{|S^+|}{|S|} \log_2 \frac{|S^+|}{|S|} - \frac{|S^-|}{|S|} \log_2 \frac{|S^-|}{|S|}, \tag{8}$$

and the information gain γ with respect to an attribute $f(\boldsymbol{x}) \leq c$:

$$\begin{aligned}\gamma(S, f(\boldsymbol{x}) \leq c) = \epsilon(S) - & \left(\frac{\left|S_{f(\boldsymbol{x}) \leq c}\right| \epsilon(S_{f(\boldsymbol{x}) \leq c})}{|S|} + \right. \\ & \left. \frac{\left|S_{\neg f(\boldsymbol{x}) \leq c}\right| \epsilon(S_{\neg f(\boldsymbol{x}) \leq c})}{|S|} \right),\end{aligned} \tag{9}$$

where $S_{f(\boldsymbol{x}) \leq c}$ and $S_{\neg f(\boldsymbol{x}) \leq c}$ are samples satisfying $f(\boldsymbol{x}) \leq c$ and samples that do not. The DT prefers to select attributes with higher information gain, i.e., the partition result that has a dominant number of samples in one certain category.

After a DT is learned, by recursively traversing the DT from its root to leaves, we can get a symbolic expression $\mathcal{L}_p$ by taking "And ($\wedge$)" on attributes along a certain path to $\top$ and "Or ($\vee$)" between such paths.

B Solution Space Analysis

In this section, we discuss what properties a solution interpretation for predicate p of a given CHC system $\mathcal{H}$ should hold. We use $U_{\mathcal{S}}$ to denote the universe set of any safe zones and positive samples, i.e., $\forall \mathcal{S}_p, \mathcal{S}_p \subseteq U_{\mathcal{S}}$ and $\forall s_p^+, s_p^+ \in U_{\mathcal{S}}$. $U_{\mathcal{U}}$ is defined similarly as the universe set of any unsafe zones and negative samples.

Lemma 11 (Disjoint $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$). *For any SAT $\mathcal{H}$, $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$ are disjoint.*

Proof. Suppose in a SAT CHC system $\mathcal{H}$, $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$ is not disjoint, i.e., there is a sample s, $s \in U_{\mathcal{S}}$ and $s \in U_{\mathcal{U}}$. According to the definition of $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$, s is both a positive and a negative sample. According to Lemma 4, $\mathcal{H}$ is UNSAT, which contradicts the assumption. Therefore, $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$ is disjoint in any SAT CHC system $\mathcal{H}$. □

Given Lemma 11, the state (i.e., sample) space can be divided into three parts shown in Fig. 5: $U_{\mathcal{S}}$ encapsulating positive samples and safe zones, $U_{\mathcal{U}}$ encompassing negative samples and unsafe zones, and an "irrelevant zone" $U_{\mathcal{I}}$ comprising samples that are neither positive nor negative.

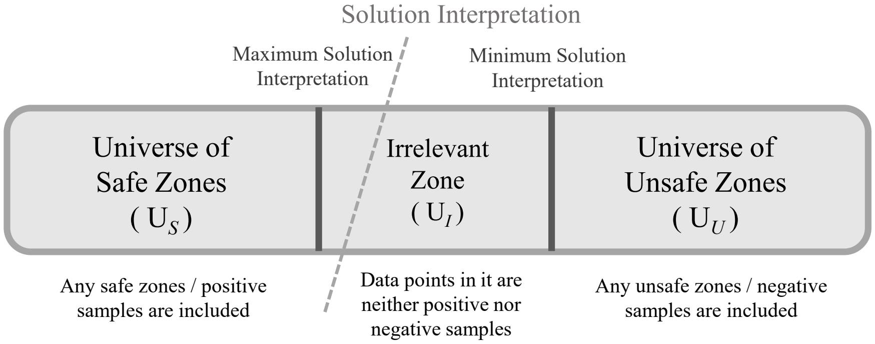
Solution Interpretation
Maximum Solution Interpretation
Minimum Solution Interpretation
Universe of Safe Zones (U$_S$)
Irrelevant Zone (U$_I$)
Universe of Unsafe Zones (U$_U$)
Any safe zones / positive samples are included
Data points in it are neither positive nor negative samples
Any unsafe zones / negative samples are included

Fig. 5: An illustration of the solution space of a CHC system in the state space. The maximum and minimum solution interpretations represent the strongest and weakest solution interpretations respectively. [17] shows the existence of them.

Lemma 12. *A solution interpretation $\mathcal{I}^* [p]$ cannot intersect with $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$.*

Proof. Suppose $\mathcal{I}^*[p]$ intersects with $U_{\mathcal{S}}$, i.e., $\exists s_p^+, \mathcal{I}^*[p](s_p^+) = \bot$. This contradicts with Definition 4. Suppose $\mathcal{I}^*[p]$ intersects with $U_{\mathcal{U}}$, i.e., $\exists s_p^-, \mathcal{I}^*[p](s_p^-) = \top$. This contradicts with Definition 5. Thus, $\mathcal{I}^*[p]$ cannot intersect with both $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$. □

We additionally answer the question of what properties a solution interpretation of $\mathcal{H}$ should hold:

Lemma 13 (Solution Space). *$\mathcal{I}$ is a solution interpretation of $\mathcal{H}$ iff:*

1) $\forall p$, $\mathcal{I}[p]$ does not intersect with $U_{\mathcal{S},p}$ and $U_{\mathcal{U},p}$;

2) For any $(n+1)$-ary implication sample $(s_1^{\rightarrow}, \cdots, s_n^{\rightarrow}, s_h^{\rightarrow})$ of body predicates $(p_1, ..., p_n)$ and head predicate h, $\mathcal{I}[p_1](s_1^{\rightarrow}) \wedge \cdots \wedge \mathcal{I}[p_n](s_n^{\rightarrow}) \rightarrow \mathcal{I}[h](s_h^{\rightarrow}) = \top$.

The Advantage of Chronosymbolic Learning. Lemma 12 and 13 suggest the importance of finding the coverage of $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$ efficiently. If $U_{\mathcal{S}}$ or $U_{\mathcal{U}}$ contains finite samples, in the worst-case scenario, traversing all states of either $U_{\mathcal{S}}$ or $U_{\mathcal{U}}$ is necessary. Symbolic zonal information, as shown in Definition 4 and 5, enables efficient computational traversal of a set of states, as learning is not required. Even when $U_{\mathcal{S}}$ and $U_{\mathcal{U}}$ contain infinite samples, zones still offer valuable information by efficiently summarizing the categories of a set of states. This enables the learner to focus on other segments of the state space that may pose challenges for the reasoner to explore.

C Implementation Details

Our artifact[31] implementing Chronosymbolic Learning is built in Python 3.10 and utilizes Microsoft Z3 [9] version 4.8.11.0 as the backend SMT solver. Our tool supports the SMT-LIB2 format and the Datalog (rule-query) format. It is compatible with CHC systems containing single or multiple predicates with arithmetic and Boolean variables.

C.1 Learner: Data-Driven CHC Solving

The *learner* module in our tool leverages an induction-based and CEGAR-inspired [8] CHC solving scheme. It has two sub-modules: the *dataset* and the *machine learning toolchain* (composed of DT and SVM, as shown in Section 2.2). The dataset stores the positive and negative samples converted from counterexamples[32] and the corresponding predicates for inductive learning. The machine learning toolchain takes the samples in the dataset as input and outputs a *partition* that can correctly classify all these samples.

[31] The artifacts are available on this anonymous link: `https://anonymous.4open.science/r/Chronosymbolic-Learning-343`

[32] See Section 4.1 for details.

SVM. We apply LIBSVM [6] as our default SVM engine. Positive and negative samples undergo iterative SVM classification until all samples are correctly categorized [41]. The resulting hyperplanes are subsequently utilized as attributes within the DT module. Boolean variables are skipped during SVM learning.

DT. We offer APIs for C5.0[33] [32] and CART (Classification and Regression Trees) [4] implemented in scikit-learn [30]. In addition to the attributes induced by SVMs, we incorporate several default attributes into DT, making the tool more efficient. These attributes encompass common patterns in inductive invariants such as the octagon abstract domain [26] ($\pm v1 \pm v2 \geq c$), as well as extensions for specific non-linear operators like `mod` and `div`. We also implement an auto-find feature capable of finding the "`mod` k" patterns in the CHC system automatically. It resembles the idea of some synthesis-based methods, which take often-appearing patterns in the CHC system as a part of the grammar [12]. We also provide a visualization tool for analyzing how DTs change through time.

Dataset. We provide two options for implementation.

A. *Collecting all data samples until the current iteration and forwarding them all to the learner.* While this approach guarantees accuracy and progress, it can become increasingly burdensome for the learner to induce a partition.
B. *Maintaining recent a positive and b negative samples in a queue.* This enhances learner efficiency but sacrifices some precision and lacks a progress guarantee. Empirically, both approaches yield nearly the same global performance.

C.2 Reasoner: Zones Discovery

Our lightweight reasoner implementation aims to enhance hypothesis generation by assisting the learner. It prioritizes simplicity in zones to minimize the computational burden on the backend SMT solver, ensuring smooth execution of the guess-and-check procedure. To maintain this simplicity, our implementation follows these principles: 1) *Initialization and Expansion*: The reasoner conducts initialization and expansion primarily at the beginning of the process. 2) *Complexity Control Criteria*: During zone expansion, specific criteria are enforced to prevent zones from becoming excessively complex: Firstly, CHCs with body constraints exceeding a length[34] of l_1 or with free variables exceeding l_2 are skipped. Secondly, a zone reaching a length exceeding l_3 triggers the termination of expansion for that zone. Lastly, a zone introducing ineliminable quantifiers also leads to the termination of the expansion procedure for that zone.

C.3 Preprocessing of CHC Systems

To handle CHC systems in various styles and formats, we preprocess the input CHC system in these aspects:

[33] https://www.rulequest.com/see5-info.html
[34] The "length" refers to the size of the internal representation of Z3.

1. Rewrite the terms in predicates to ensure that the predicates are only parameterized by distinctive variables (such as $p(v_0, v_1, v_2)$);
2. Clear and propagate predicates that are $\top$ or $\bot$;
3. Simplify the inner structure of each CHC if possible.

D Experimental Details

We provide more details in Section 5 as follows. The detailed running logs and timing statistics area available in our artifact[35].

D.1 Statistics about the benchmarks

Most benchmarks have less than 10 rules, 5 predicates, and 10 variables for each predicate. On average, on all benchmarks under the Chronosymbolic-single setting, we need 1.64 rounds of outer while loop in Alg. 1, line 5, and 243.2 iterations of for loop at line 8, so 243.2 counterexamples are generated on average.

For quantifier elimination (QE), empirically, in most cases, we can do QE when expanding the zones (more than 90% of cases). However, as described before, considering not slowing down the backend solver, the expansion stops when we cannot do QE.

For non-linear arithmetic benchmarks, we have 19 in total. Aside from them, GSpacer achieves 219/269, and Chronosymbolic-cover achieves 237/269.

In the experiment, 3 exclusive instances are solved in the Chronosymbolic-single setting (compared with our baselines)[36].

D.2 Detailed Settings for Chronosymbolic-cover

The Chronosymbolic-cover setting mainly covers the following experiment:

1. Different expansion strategy of zones (e.g., do not expand at all, small or large limit of size, etc.);
2. Different dataset configuration (e.g., whether to enable queue mode on the positive and negative dataset, the length of the queue, how we deal with tentative data samples);
3. Different strategies for scheduling the candidate hypothesis described in Section 1;
4. Different DT settings in Appendix C.1.

[35] `https://anonymous.4open.science/r/Chronosymbolic-Learning-343A/experiment/README.md`

[36] An example case is shown at: `https://anonymous.4open.science/r/Chronosymbolic-Learning-343A/examples`

D.3 Results on Using Different Random Seeds in DT

We further examine how randomness affects our system. In this section, we use six different random seeds $(1, 13, 137, 400, 1371, 13711)$ in CART for each experiment. We apply dataset option B) in Section 4.2 for simplicity in analysis. Other configurations remain the same as "Chronosymbolic-single" as described in Section 5.1. As an example, we only consider the safe instances in our test suite.

Table 6: Performance evaluation across different random seeds. "#solved" and "percentage" stands for solved instances among 235 safe instances. "avg-time" is the average time consumed on each instance (including timeout or crashed ones). "And" stands for the instances solved under all random seed configurations, while "Or" for the instances solved under at least one configuration.

random seed	#total	percentage	avg-time (s)
1	183	77.87%	89.04
13	183	77.87%	89.89
137	177	75.32%	94.07
400	186	79.15%	83.19
1371	188	80.00%	86.52
13711	183	77.87%	84.87
And	175	74.47%	-
Or	195	82.98%	-

As shown in Table 6, the result that our approach consistently solves a substantial number of instances across various random seed configurations demonstrates the robustness of our approach. Different random seeds yield varying partition results for the same dataset, leading to diverse hypotheses exploring distinct directions within the solution space. The "Or" result highlights how different seed configurations can collectively cover a broader range of instances.