Multi-Phase Invariant Synthesis

Daniel Riley
Florida State University
United States
driley@cs.fsu.edu

Grigory Fedyukovich
Florida State University
United States
grigory@cs.fsu.edu

ABSTRACT

Loops with multiple phases are challenging to verify because they require disjunctive invariants. Invariants could also have the form of implication between a precondition for the phase and a lemma that is valid throughout the phase. Such invariant structure is however not widely supported in state-of-the-art verification. We present a novel SMT-based approach to synthesize implication invariants for multi-phase loops. Our technique computes Model Based Projections to discover the program's phases and leverages data learning to get relationships among loop variables at an arbitrary place in the loop. It is effective in the challenging cases of mutually-dependent periodic phases, where many implication invariants need to be discovered simultaneously. Our approach has shown promising results in its ability to verify programs with complex phase structures. We have implemented and evaluated our algorithm against several state-of-the-art solvers.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**.

KEYWORDS

automated safety verification, inductive invariant synthesis, satisfiability modulo theories, model based projection

ACM Reference Format:
Daniel Riley and Grigory Fedyukovich. 2022. Multi-Phase Invariant Synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549166

1 INTRODUCTION

Automated software verification tools often delegate their computational tasks to solvers for Satisfiability Modulo Theories (SMT) and *Constrained Horn Clauses* (CHC). The latter aims at synthesizing inductive invariants for loops and recursive calls, and it enables sound reasoning about software safety. Existing CHC solvers extensively rely on SMT solvers. The ability to discover a solution often critically depends on the capabilities of the SMT solver to apply interpolation or quantifier elimination. Recently proposed *guess-and-check* methods [18, 50, 54], while still relying on SMT solvers to prove invariants, are guided by external sources, such as user-provided grammars, templates, or concrete data to generate different pieces of invariants.

Programs that display control-flow divergence in the bodies of their loops usually require *multi-phase invariants* that are usually disjunctive and are more difficult to infer [5, 46, 56, 57]. Our work is motivated by the need for improved methods to verify multi-phased systems used in several fields, specifically:

- Control software and reactive systems perform tasks depending on their environment [3, 35, 46]. Such changes boil down to phases, where the behavior of the system will be different based on the conditions. These systems are often safety critical such as self-driving systems, flight software, or medical diagnostic software.
- Multi-phase loops may arise from a common transformation technique such as loop flattening [42], used by verification frontends when translating from a programming language such as C or Java into CHCs [28]. Verification systems have become more modular, with solvers in the backend relying on frontend systems to translate a given program. Consequently the backend solvers do not always have control over the result of the translation and can be given multi-phase loops even if the source program does not have them.
- Termination checking is often reduced to safety verification [11, 21, 40]. A ranking function is found to overestimate the number of iterations of a loop. In a complicated case of lexicographic ranking function synthesis [12, 40, 62], various program phases need to be analyzed. Multi-phase invariants may help in these cases.

Our new approach to multi-phase invariant synthesis aims to find predicates that have the form of implication, where the left-hand side is called a *phase guard* uniquely describing when a certain phase is enabled, and the right-hand side is called a *phase lemma*. Implications by definition encode a disjunction, so their use here intuitively captures the disjunctive nature of multi-phase loops. Generation of phase lemmas is challenging because they should be valid under their phase guards. In this paper we describe a new algorithm to derive both phase lemmas and their phase guards automatically.

We build on top of the Houdini [23] strategy, where invariants are selected from a set of candidates. The synthesis loop can be parameterized by grammars, be data-driven, or follow some semantic inference rules. It begins similarly to many traditional algorithms, i.e., by guessing candidates and checking their inductiveness using an SMT solver. Our main idea is to synthesize a phase guard for each candidate that fails the inductiveness check, thus, weakening it. Intuitively, a phase guard represents a set of states, for which the failed candidate is inductive.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549166

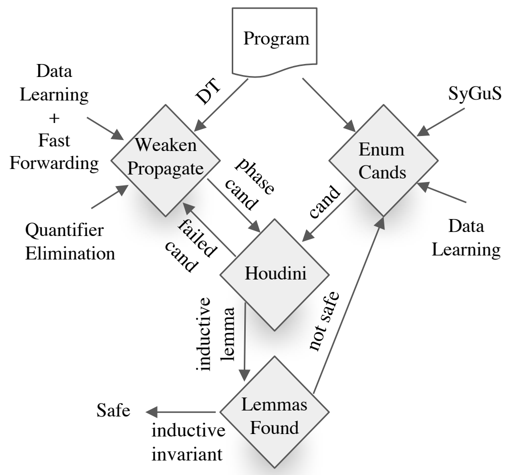

Figure 1: Architecture of ImplCheck.

Our first main insight is that phase guards are computed by a Model-Based Projection (MBP) [7, 37] that under-approximates quantifier elimination (QE) over the loop body. Because a diverging control-flow can be represented as a disjunctive formula, its under-approximation (a disjunct) encodes a loop phase. We use an MBP algorithm to lazily [45] compute all phase guards and organize them in a decision tree.

The second important design choice is the use of data traces from a particular phase for discovering phase lemmas. For numeric programs, existing methods [19, 38, 47, 60] have already proven their success, but they critically depend on relevant data. We propose a new method for data collection using an SMT solver that takes a bounded unrolling of a phase beginning at a state where the phase guard does not hold and proceeds to states where it holds. This avoids the creation of the full unrolling that executes the loop from its initial state up until the entry to the current phase. However, a naive application of this idea may result in some important computation from a previous phase not being taken into account. To mitigate this, our algorithm reuses the invariants from the previous phases and any global invariants[1], if they have already been found.

The high-level overview of the approach is shown in Fig. 1. The algorithm takes a program encoded in CHCs, then extracts and organizes its phases into a decision tree. The search of candidates begins with an enumerative approach that supplies an initial set of candidates, which, if they fail the Houdini check, are passed to our weakening engine. The enumeration block of our framework is parameterizeable by any paradigm which synthesizes expressions, including a Syntax-Guided Synthesis [2] (SyGuS) approach and/or a data learner [60] that extracts invariants from bounded behaviors. In Weaken-and-Propagate, candidates are synthesized into the form of implication, with the premise being a phase guard and the conclusion being a phase lemma. These implication candidates are passed to Houdini to calculate their inductive subset. When the collection of Lemmas returned by Houdini is strong enough to block the error states, the program is verified safe.

[1]A global invariant is one that is not associated with a *phase guard*.

Our new algorithm has been implemented in a tool called ImplCheck on top of the FreqHorn [18] CHC solver and the Z3 [14] SMT solver and uses their quantifier elimination, MBP, and data learning algorithms. We have evaluated it against state-of-the-art on a set of public benchmarks. Of particular interest are a subset from CHC-COMP, that include a variety of phase structures within their loops, which we believe tests the robustness of our approach. The set of experiments confirms that ImplCheck is capable of inferring a much larger set of multi-phase invariants than the competitors and that its overhead while running on single-phase benchmarks is small.

The rest of the paper is structured as follows. We proceed with an illustrating example in Sect. 2 and a brief background in Sect. 3. Sect. 4 summarizes our contributions on synthesizing phase guards using MBP and organizing them in a decision tree. The overview of the algorithm is then presented in Sect. 5. Sect. 6 gives two strategies on synthesizing phase lemmas: by propagation using quantifier elimination (a general approach), and accelerated data-learning (an approach designed for numeric programs). The implementation is briefly reported in Sect. 7, and the evaluation and comparison against the state-of-the-art is detailed in Sect. 8.

2 ILLUSTRATING EXAMPLE

The program in Fig. 2 illustrates the main concepts of our approach. It has three variables: `x` always increments, `y` is assigned nondeterministically, and `z` increments conditionally. The loop body has a single conditional and up to three different phases, depending on the value of `y`. The plot in Fig. 2 depicts the case when `y` is positive, and thus the loop goes to three phases: 1) from the initial state until the guard starts to hold, 2) when the guard holds, and 3) when the guard does not hold anymore. After the loop terminates and if `y` is positive and `x` is sufficiently large, our goal is to prove that `z` equals `1000`.

A safety property can be proved by finding an inductive invariant which intuitively 1) is implied by the initial states formula, 2) is closed under the transition relation, and 3) implies the safety property. An inductive invariant for our illustration program is given in Fig. 5. It is nontrivial: in our experiments, *ten state-of-the-art invariant synthesizers* failed to discover it automatically within a one hour timeout. Intuitively, the invariant describes a relation between variables `x`, `y`, and `z` in three phases of the loop when `y` is positive: if `x` is sufficiently small then `z` remains at zero, then `z` and `x` grow simultaneously until at some point `z` reaches `1000` and stagnates. Our algorithm generates an individual invariant (called a *lemma*) for each of the three phases and then conjoins them together. Each lemma has the form of an implication, its antecedent is called a *phase guard*, and its consequent is called a *phase lemma*.

To discover phase guards, our approach analyzes the control flow of the program and organizes its phase-reachability information in a *decision tree* (DT), shown in Fig. 3. DT approximates the order and conditions of visiting all possible phases. That is, if `y` is positive, then $y > x \texttt{ div } 1000$ holds at the beginning, $y = x \texttt{ div } 1000$ holds after some iterations, and $y < x \texttt{ div } 1000$ holds at the end (and thus the rightmost branch of the DT has depth three). In general, we can show that every program has only a finite number of phases,

```
int x, y, z;
x, z = 0;
y = nondetInt();
while (*) {
  if (y == x/1000)
    z++;
  x++;
}
if (y > 0 &&
    x > 1000*(y + 1))
  assert(z == 1000);
```

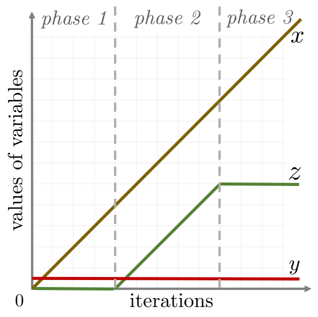

Figure 2: C-like program with a multi-phase loop.

$$\begin{tikzcd}
 & \top \arrow[dl] \arrow[d] \arrow[dr] & \\
y < 0 \land y < x \texttt{ div } 1000 & y = 0 \land y = x \texttt{ div } 1000 \arrow[d] & y > 0 \land y > x \texttt{ div } 1000 \arrow[d] \\
 & y = 0 \land y < x \texttt{ div } 1000 & y > 0 \land y = x \texttt{ div } 1000 \arrow[d] \\
 & & y > 0 \land y < x \texttt{ div } 1000
\end{tikzcd}$$

Figure 3: Decision tree for phase-guard selection.

First phase			Second phase			Third phase		
x	y	z	x	y	z	x	y	z
0	5	0	5001	5	1	6000	5	1000
1	5	0	5002	5	2	6001	5	1000
2	5	0	5003	5	3	6002	5	1000

Figure 4: Data matrices capturing variable values.

$$\begin{aligned}
&Inv \mapsto \lambda x, y, z\,.\\
&\quad y > 0 \land y > x \texttt{ div } 1000 \implies z = 0 \land\\
&\quad y > 0 \land y = x \texttt{ div } 1000 \implies z = x - 1000y \land\\
&\quad y > 0 \land y < x \texttt{ div } 1000 \implies z = 1000
\end{aligned}$$

Figure 5: Safe inductive invariant.

precomputed before the analysis, so our algorithm can cheaply traverse the DT multiple times throughout the verification.

To discover phase lemmas, we can in principle follow various techniques, from guess-and-check to fixpoint computation. However, for numeric programs, the most effective approach in our experience is based on data learning and Houdini. It begins with constructing a data matrix for all the phases individually (shown in Fig. 4) and inferring relationships between variables. For the first phase, the process is straightforward: a value for y is randomly picked: and if it is positive then we have to follow the rightmost branch of the DT. Thus, the values for x and z in the first three iterations are inferred precisely, giving us a phase lemma $z = 0$ to be associated with the phase guard $y > 0 \land y > x \texttt{ div } 1000$. Then, the chosen branch of the DT leads us to the second phase (i.e., in which $y > 0 \land y = x \texttt{ div } 1000$ holds). The data matrix creation in this case is trickier since for $y = 5$ we would have to unwind the loop 5000 times (which is expensive). Instead, we simulate this using a technique called *fast-forwarding*: the loop unrolling begins at an arbitrary iteration, such that the first phase guard holds at the beginning, but the second guard holds at the end. For our example, it is sufficient to get the second data matrix and consequently the second phase lemma $z = x - 1000y$. Lastly, the third phase is computed similarly, and the conjunction of them is sufficient for proving the safety property in the program.

3 BACKGROUND

This paper approaches the problem of automated software verification by reduction to *Satisfiability Modulo Theories* (SMT) problems.

3.1 Logic Notation and Main Routines

Automated SMT solvers determine the existence of a satisfying assignment to variables (also called a *model*) of a first-order logic formula. We write $\mathcal{M} \models \varphi$ to denote that a model $\mathcal{M}$ satisfies a formula φ (and $\exists \mathcal{M} \models \varphi$ to denote the satisfiability of φ). Formula φ is logically stronger than formula ψ (denoted $\varphi \implies \psi$), if every model of φ also satisfies ψ. The unsatisfiability of formula φ is denoted $\varphi \implies \mathit{false}$. By writing $\psi(x)$, we denote a predicate over free variables x. We use *ite* to denote *if-then-else*.

For a formula φ, terms/formulas a and b, we write $\varphi[b/a]$ to denote φ after all instances of a are replaced by b. For a set of terms/-formulas X and a mapping $\mathcal{M}$ from X to other terms/formulas, $\varphi[\mathcal{M}/X]$ denotes the simultaneous replacement of all $x_1, x_2, \ldots \in X$ by $\mathcal{M}(x_1), \mathcal{M}(x_2), \ldots$, respectively.

By Ψ we denote the space of all possible quantifier-free formulas in our background theory and by *Vars* a sequence of possible variables. Because in the paper we mainly deal with conjunctive formulas (which are created by adding/dropping conjuncts), we sometimes slightly abuse the notation and refer to $S \in 2^{\Psi}$ as a conjunction of all its elements, i.e., $\bigwedge\limits_{c \in S} c$.

The problem of quantifier elimination is for a given $\exists \vec{y}\,.\,\psi(\vec{x}, \vec{y})$ to generate an equivalent $\vec{y}$-free formula, and it can be approached by converting $\psi(\vec{x}, \vec{y})$ to DNF, eliminating quantifiers from each disjunct, and disjoining the results. Our main insight, instead, is to build the phase guard generation engine on top of a notion of Model-Based Projection (MBP) [7] that under-approximates QE without converting to DNF. Specifically, one could compute a number of MBPs lazily, and disjoin them. The number of MBPs is in practice significantly lower than the number of disjuncts in the DNF of the same formula, which yields significant performance gains.

Definition 3.1. Given a formula ψ over $\vec{x}, \vec{y}$, and a model $\mathcal{M}$, an $MBP_{\vec{y}}(\mathcal{M}, \psi)$ is a $\vec{y}$-free formula if the following hold:

$$\text{if } \mathcal{M} \models \psi(\vec{x}, \vec{y}) \text{ then } \mathcal{M} \models MBP_{\vec{y}}(\mathcal{M}, \psi)$$
$$MBP_{\vec{y}}(\mathcal{M}, \psi) \implies \exists \vec{y}\,.\,\psi(\vec{x}, \vec{y})$$

For linear integer arithmetic, an MBP for formula ψ and its model $\mathcal{M}$ can be constructed from literals of formula ψ, converted to the Negation Normal Form (NNF), finding literals that evaluate to true on $\mathcal{M}$, and eliminating *Vars′* from their conjunction. More formally, it is presented in Algorithm 1. We defer an example of the algorithm run to Sect. 4.1, where it is used in the context of multi-phase program verification.

Algorithm 1: $\text{MBP}_{Vars'}(\mathcal{M}, \varphi)$: basic MBP construction

```
    Input: $\mathcal{M}$: model, $\varphi(Vars, Vars')$: formula
    Output: $\psi$: $Vars'$-free MBP
1   $\varphi \leftarrow \text{TONNF}(\varphi)$;
2   $\psi(Vars, Vars') \leftarrow \{\ell \mid \ell \in \text{LITERALS}(\varphi) \wedge \mathcal{M} \models \ell\}$;
3   return $\text{QE}(Vars', \exists Vars' \,.\, \psi(Vars, Vars'))$;
```

3.2 Programs and Transition Systems

We view programs as *transition systems* and throughout the paper use both terms interchangeably[2].

Definition 3.2. A *transition system* P is a tuple $\langle Vars \cup Vars', Init, Tr\rangle$, where $Vars$ and $Vars'$ are copies of the sequence of variables at the beginning and the end of a transition, respectively; $Init$ and Tr are the symbolic encodings of the *initial states* and the *transition relation.*

A program P and an encoding Bad of *error states* define a *verification problem*, which is satisfiable if the set of error states is unreachable. If satisfiable, a solution of a verification problem is a *safe inductive invariant*, represented by a formula Inv such that: Inv over-approximates $Init$, is closed under Tr, and Inv is strong enough to block all error states.

Definition 3.3. Given $P = \langle Vars \cup Vars', Init, Tr\rangle$; a formula Inv is a *safe inductive invariant* if the following conditions (respectively called an initiation, a consecution, and a safety) hold:

$$Init(Vars) \implies Inv(Vars) \quad (1)$$
$$Inv(Vars) \wedge Tr(Vars, Vars') \implies Inv(Vars') \quad (2)$$
$$Inv(Vars) \wedge Bad(Vars) \implies false \quad (3)$$

Example 3.4. The problem of finding an inductive invariant for program in Fig. 2 can be formulated as follows.

$$\begin{aligned} x = 0 \wedge z = 0 &\implies Inv(x,y,z) \\ Inv(x,y,z) \wedge x' = x+1 \wedge y' = y \wedge & \\ z' = ite(y = x \texttt{ div } 1000, z+1, z) &\implies Inv(x',y',z') \\ Inv(x,y,z) \wedge y > 0 \wedge & \\ x > 1000(y+1) \wedge \neg(z \geq 1000) &\implies false \end{aligned}$$

3.3 Data Learning

We use data learning to discover inductive invariants by examining program traces. Traces are gathered as a finite unrolling based on a Bounded Model Checking (BMC) formula [6].

An *unrolling* of length m of a program $P = \langle Vars, Init, Tr\rangle$ is a conjunction:

$$unrl(Vars, \ldots, Vars^{(m)}) \stackrel{\text{def}}{=}$$
$$Init(Vars) \wedge Tr(Vars, Vars') \wedge \ldots \wedge Tr(Vars^{(m-1)}, Vars^{(m)})$$

with each i, j, k, n, such that $i \neq j$ and $k \neq n$, $Vars^{(i)}[k] \neq Vars^{(j)}[n]$.

Note that we rely on sequences (rather than on sets) of variables $Vars$, and we can extract the nth variable of $Vars$ via $Vars[n]$. This is useful for obtaining data to learn predicates, which consist of values for each variable in each iteration of the program. It can be either obtained by a dynamic execution, or extracted from a model, $\mathcal{M} \models unrl(Vars, \ldots, Vars^{(m)})$, for some unrolling of the program. The values obtained from the model are then stored in a matrix.

An $m \times n$ matrix A is composed of m rows representing the iterations of a program and of n columns representing the value of a variable $Vars[j]$. Each element, $A[i,j]$, of A holds the value in the i^{th} iteration of the j^{th} variable, that is, if $\mathcal{M} \models unrl(Vars, \ldots, Vars^{(m)})$ then $A[i,j] \stackrel{\text{def}}{=} \mathcal{M}(Vars^{(i)}[j])$.

Fig. 4 shows three example matrices representing three different phases of Example 3.4. An unrolling for our motivating example begins at the initial state and captures the information shown in the **First Phase** matrix of Fig. 4. For instance, the first matrix is computed from a model of $unrl(Vars, Vars', Vars'')$, contained of $Init$ and two copies of Tr. After an unrolling, data learning proceeds to analyze the resulting matrix.

From linear algebra, given a vector space $\mathbf{V}$ over a field $\mathbf{F}$, its *basis* $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_i\}$ is the minimal subset of $\mathbf{V}$ if every vector $v \in \mathbf{V}$ can be written as a linear combination of $\mathbf{B}$. The basis $\mathbf{B}$ of the null space, the set of all vectors that when multiplied by A equal to $\vec{0}$, of A produces candidate invariants [60]. Invariants found from basis vectors take the shape of equalities, $a_1x_1 + a_2x_2 + \ldots a_ix_i = 0$.

Example 3.5. Let A be the matrix of **Second Phase** in Fig. 4. Solving $A\vec{b} = 0$, the solution for $\vec{b}$ is $\begin{pmatrix} -1 \\ 1000 \\ 1 \end{pmatrix}$. Then the candidate invariant is $z = x - 1000y$.

A weakness of this approach stems from the construction of the data matrix used to produce candidate invariants for an arbitrary phase – it would require constructing potentially large unrollings. We will explore a way around this weakness in Sect. 6.

3.4 Inductive Subset Extraction

In this paper, we target invariants composed from multiple *lemmas* i.e., $Inv = \ell_0 \wedge \ldots \wedge \ell_n$, where each ℓ_i passes the initiation and consecution checks from Def. 3.3. Thus, invariants can be found by an enumeration of candidate formulas and finding a subset of them, such that their conjunction fulfills the third (safety) check as well. Given an initial set of formulas (either captured in a predetermined grammar [50, 54], obtained from syntax [18], or behaviors [19, 47]), an enumerative approach aims at gradually narrowing it to a subset the conjunction of which is a solution. The Houdini algorithm, shown in Algorithm 2, finds formulas that pass the initiation and then continues to iteratively remove bad candidates using so called *counterexamples-to-induction.* If the remaining candidates are not consistent with the set of error states, the program is safe.

Example 3.6. Let $Cands = \{z = 0, z \geq 0\}$ for Example 3.4, then Algorithm 2 iterates two times. First it finds that $z = 0$ is not inductive, e.g., by finding model $\mathcal{M} = \{x \mapsto 5000, y \mapsto 5, z \mapsto 0, x' \mapsto 5001, y' \mapsto 5, z' \mapsto 1\}$, and it keeps only $z \geq 0$ because $\mathcal{M}(z') \geq 0$. Second, it finds that $z \geq 0$ is inductive (and returned as a lemma), which is, however, not enough for proving the safety.

While the algorithm is useful for finding conjunctive invariants, its weakness is in a too aggressive weakening (i.e., keeping or dropping the entire candidate). However, the dropped candidate

[2] Although the presentation assumes single-loop programs, our implementation works also for programs with multiple loops.

```
Algorithm 2: HOUDINI($P, Bad, Cands$), cf. [23]
  Input: $P = \langle Vars \cup Vars', Init, Tr\rangle$: program; $Bad$: error states
         $Cands \in 2^{\Psi}$
  Output: $res \in \langle$SAFE, UNKNOWN$\rangle$, set $Lemmas \subseteq Cands$: of
          inductive lemmas
1 $Cands \leftarrow \{c \in Cands \mid Init(Vars) \implies c(Vars)\}$;
2 while $\exists \mathcal{M}$, s.t.
   $\mathcal{M} \models Cands(Vars) \land Tr(Vars, Vars') \land \neg Cands(Vars')$ do
3     $Cands \leftarrow \{c \in Cands \mid \mathcal{M} \models c(Vars')\}$;
4 if $Cands(Vars) \land Bad(Vars) \implies false$ then
5     return SAFE, $Cands$;
6 return UNKNOWN, $Cands$;
```

$z = 0$ can be weakened in another way, by adding a phase guard, as in Fig. 5. Our novel weakening strategy is discussed in Sect. 5.

4 PHASE REACHABILITY TREE

The reason for control-flow divergence can often be extracted directly from transition relations[3]. When a transition relation is in a disjunctive normal form (DNF), then each disjunct represents a group of program executions belonging to the same phase. For deterministic programs, these disjuncts have conditions on the states at the beginning of a transition (usually called *source* states), that has to be true for all executions in the corresponding phase. These conditions are quite important for our invariant synthesis since they can be used to weaken candidates that do not pass a HOUDINI run, and thus we call them *phase guards*. Obtaining phase guards in our approach proceeds in two stages. First, our algorithm computes a finite number of phases from the transition relation using quantifier elimination (QE). The resulting formulas represent phase reachability information that are arranged in a decision tree (DT), and lastly strengthened based on their position in the DT to reflect some context-specific information.

4.1 Model-Based Projections as Phase Guards

Informally, a guard is a description of a subset of source states that need to be true for all transitions of the phase. Instead of converting a transition relation to DNF, and then applying QE to get a guard for each disjunct, our approach picks one representative transition of a phase and identifies its phase guard. A transition is represented by a model of the Tr formula over source and destination variables $Vars$ and $Vars'$. Thus, we have to compute an MBP of formula $\exists Vars' \,.\, Tr(Vars, Vars')$ based on that model.

Example 4.1. Recall Example 3.4, in which the transition relation is as follows.

$$x' = x + 1 \land y' = y \land z' = ite(y = x \texttt{ div } 1000, z + 1, z)$$

[3]The main benefit of conducting an analysis over this symbolic encoding (vs over the program's source code) is that the approach is language-agnostic. Furthermore, is is also insensitive to the verification frontend that does the encoding. Specifically, tools often perform various program transformations, including ones that change the control-flow structure, sometimes drastically. Thus, the phase reachability information obtained from the source-code level may not be adequate at the level of the symbolic encoding.

```
Algorithm 3: ALLMBP$_{Vars'}(\mathcal{M}, \varphi)$
  Input: $\varphi(Vars, Vars')$: formula
  Output: MBPs: $Vars'$-free set of Model-Based Projections
1 MBPs $\leftarrow \varnothing$;
2 while $true$ do
3     if $\exists \mathcal{M} \models \varphi(Vars, Vars') \land \bigwedge_i \neg \mathrm{MBPs}_i(Vars)$ then
4         MBPs $\leftarrow$ MBPs $\cup \{\mathrm{MBP}_{Vars'}(\mathcal{M}, \varphi(Vars, Vars'))\}$;
5     else
6         return MBPs;
```

The formula is satisfied by the model $\mathcal{M} = \{x \mapsto 0, y \mapsto 1, z \mapsto 0, x' \mapsto 1, y' \mapsto 1, z' \mapsto 0\}$, thus we proceed to generating an MBP. Algorithm 1 then splits the transition relation into literals:

$$\text{LITERALS}(Tr) = \{x' = x + 1, z' = z + 1, z' = z, y' = y,$$
$$y < x \texttt{ div } 1000, y = x \texttt{ div } 1000, y > x \texttt{ div } 1000\}$$

It is easy to see that after collecting literals that are satisfied by $\mathcal{M}$ and conjoining them, we can get an under-approximation of the transition relation that only describes the first phase:

$$y > x \texttt{ div } 1000 \land x' = x + 1 \land z' = z \land y' = y$$

Lastly, by eliminating x', y', and z', we get the first phase guard $y > x \texttt{ div } 1000$.

To obtain all of the guards, we repeat the process of gathering models of Tr that are not covered by the previously generated guards, as presented in Algorithm 3.

4.2 Organizing Phase Guards in a Decision Tree

It is convenient to represent phases in a decision tree, that is computed exactly once, prior to the invariant synthesis run. We introduce the notion of a phase-reachability tree to approximate all possible execution scenarios, assuming that the phase guards are already computed. Intuitively, it gathers sequences of phases that could be visited by a program: the feasibility of every transition is checked by an SMT solver using the symbolic encoding of P.

Definition 4.2. Given a program $P = \langle Vars \cup Vars', Init, Tr\rangle$, a *phase-reachability tree* DT_P is a quadruple $\langle V, E, r, p\rangle$ of sets of vertices from some set V, edges $E \subseteq V \times V$, a root $r \in V$, and a vertex-labeling function $p : V \to \Psi$ such that:

- for all $\langle r, v\rangle \in E$: formula $Init(Vars) \land p(v)(Vars)$ is satisfiable,
- for all $\langle v_1, v_2\rangle \in E$, where $v_1 \neq r$:
  $p(v_1)(Vars) \land Tr(Vars, Vars') \land p(v_2)(Vars')$ is satisfiable,
- for all paths $\pi = \langle r, v_1, \ldots, v_n\rangle$ in DT_P, for every two vertices, $v_i, v_j \in \pi$: formulas $p(v_i)$ and $p(v_j)$ are equisatisfiable only if $i = j$.

Every path π in DT_P corresponds to (a prefix) of some execution of P, possibly spurious. While the DT_P gives us incomplete information about program traces, it is often sufficient to derive phase invariants, and it is cheap to compute. Note that getting the most precise phase-reachability information for a program could be difficult (even impossible) since it may require knowing some auxiliary (helper) invariants. However, even a coarse DT_P fits well for the purposes of our approach, which reasons automatically in terms

Algorithm 4: $\textsc{StrenDT}(P, DT_P)$

```
Input: $P = \langle Vars \cup Vars', Init, Tr \rangle$: program,
       $DT_P = \langle E, V, r, p \rangle$: phase-reachability tree
Output: $DT_P$: augmented phase-reachability tree
1 for $\langle u, v \rangle \in E$ do
2     if $u = r$ then
3         $\varphi \leftarrow \textsc{abduce}(Init(Vars) \implies p(v)(Vars))$;
4     else
5         $\varphi \leftarrow \textsc{abduce}(p(u)(Vars) \land Tr(Vars, Vars') \land$
            $\neg p(u)(Vars') \implies p(v)(Vars'))$;
6     if $\varphi \not\Longrightarrow false$ and
        $\varphi(Vars) \land Tr(Vars, Vars') \implies \varphi(Vars')$ then
7         for $w \in subtree(DT_P, v)$ do
8             $p(w) \leftarrow p(w) \land \varphi$;
```

of batches of loop iterations and discovers some "locally inductive" facts about phases (see Sect. 5).

We slightly abuse the notation, and for an edge $\langle u, v \rangle$, we refer to the phase associated with u (resp. v) as to a "parent" (resp. "child") phase. Construction of the DT_P from the set of phase guards G returned by Algorithm 3 is rather straightforward: it poses a number of SMT checks, as shown in Def. 4.2, given a set of phase guards and the symbolic encoding of P. At each "parent" phase v_i and each potential "child" phase v_j from some $G_i \subseteq G$, the reachability is checked by posing a satisfiability query to an SMT solver. If successful, edge $\langle v_i, v_j \rangle$ is added to E, and the construction process recurses for v_j and $G_i \setminus \{v_j\}$. Unsatisfiability of some reachability formula guarantees that it is impossible in general to have a phase v_j right after v_i.

Note that by construction, some elements of the initial G may appear in the final DT_P several times, but only once per path. That is, each path π captures only phase guards without repetitions: even if a phase of P is visited multiple times during π, the phase guard occurs in π exactly once. In the case of interleaving phases, traversal of the DT_P "cycles" through π. Thus, the constructed DT_P always has a finite depth which makes it convenient for the further synthesis process.

4.3 Context-Specific Phase Guard Strengthening

While our primary source of phase guards is the MBP procedure, in some cases, the discovered formulas are not strong enough to be phase guards. Our motivating example illustrates this case. Recall Example 4.1, in which formula $y > x \texttt{ div } 1000$ is generated. However, if we use it solely (i.e., without conjoining with $y > 0$) in the phase guard for some invariants, our algorithm will not be able to produce the desired invariants, as in Fig. 5.

Intuitively, phase guard strengthening is needed to bring more context information into the phases, e.g., if a value of some variable(s) is unknown at the initial state, then the loop may have several different phase scenarios, depending on the ranges of possible values of that variable(s). In order to bring this specific phase-reachability information to invariants, we perform *strengthening* of DT_P after it is generated from MBPs in Sect. 4.2.

Algorithm 4 gives pseudocode of the algorithm. It has two parts: discovery of strengthening, and its propagation to the subtree. The algorithm traverses the tree *top-to-bottom* and finds edges, where the end a "parent" phase does not imply the beginning of the "child" phase. The main idea is to infer a (reasonably weaker) condition under which the "child" phase is reachable, and then add it to the phase guard of the "child phase". This is commonly achieved by abduction [15], and our pseudocode uses the following implementation based on quantifier elimination:

$$\textsc{abduce}(A(Vars) \implies B(Vars)) \overset{\text{def}}{=} \mathrm{QE}(\forall Vars \setminus W \,.\, A \implies B)$$

where W is a subset of $Vars$, which can be found heuristically. In our implementation, we enumerate different W until abduction results in a formula, that is non-trivial and locally inductive. Then it is conjoined to all phase guards associated with the vertices in the subtree.

Example 4.3. For Example 4.1, the need of strengthening of the phase guard is revealed by solving the validity of formula:

$$x = 0 \land z = 0 \implies y > x \texttt{ div } 1000$$

Because the implication does not hold, we aim at finding a (reasonably weaker) predicate φ to be conjoined with the antecedent to make the implication valid. Because there is only one model for formula $x = 0 \land z = 0$, then constraining x or z makes no sense. Thus, the abducible predicate φ has to be applied to variable y, and it can be found by QE over the following formula:

$$\neg \mathrm{QE}(\exists x, z \,.\, (x = 0 \land z = 0 \land y \leq x \texttt{ div } 1000)).$$

The resulting formula $y > 0$ is locally inductive,

Strengthening produced this way is not unique, and in principle this procedure may be repeated several times, that can further be exploited by several (maybe, parallel) runs of the main algorithm (see the next section). It is also possible to infer strengthening of DT_P beginning from the query in the reverse order. We omit these extensions of our approach in the interest of saving space.

5 INVARIANT SYNTHESIS BASED ON PHASE REACHABILITY TREE

In this section, we present our core contribution: an automated algorithm to discover multi-phase invariants. It relies on the phase-reachability tree DT_P, precomputed (and possibly strengthened) as shown in Sect. 4. The main insights are to search for candidates, that do not pass the initiation/inductiveness checks in Houdini, to attempt to weaken them using phases from DT_P, and (importantly), to *generate new candidates* for the next phases. While this section focuses on conveying the main ideas, the two important design choices for new candidate generation are deferred to Sect. 6.

5.1 Overview

Algorithm 5 gives an overview of our approach. It takes as input a set of candidates for invariants, which can be originated from any external source like data learning (recall Sect. 3.3), SyGuS, e.g. [18, 50, 54], or any set of predetermined templates.

The algorithm then invokes Algorithm 2 to compute an inductive subset of lemmas from the given set of candidates. If the conjunction of lemmas is safe, the program is correct, and the algorithm

Algorithm 5: ImplCheck(P, $Cands$, $PhaseInvs$, DT_P).

```
Input: $P = \langle Vars \cup Vars', Init, Tr\rangle$: program; $Cands \in 2^{\Psi}$,
       $PhaseInvs \in 2^{\Psi}$, $DT_P$: precomputed
       phase-reachability tree
Output: inductive $Lemmas \in 2^{\Psi}$
1  $\langle res, Lemmas\rangle \leftarrow$ Houdini($P$, $Cands \cup PhaseInvs$);
2  if $res$ = SAFE then return SAFE;
3  if $Cands \setminus Lemmas = \varnothing$ then return UNKNOWN;
4  pick $\ell \in Cands \setminus Lemmas$;
5  $PhaseInvs \leftarrow$
     WeakenAndPropagate($P, \ell, DT_P$) $\cup$ $PhaseInvs$;
6  return ImplCheck($P$, $Cands \setminus \{\ell\}$, $PhaseInvs$, $DT_P$);
```

Algorithm 6: WeakenAndPropagate(P, ℓ, DT_P).

```
Input: $P = \langle Vars \cup Vars', Init, Tr\rangle$: program; $\ell \in \Psi$: an
       invariant candidate; $DT_P = \langle V, E, r, p\rangle$:
       phase-reachability tree
Output: $Cands \in 2^{\Psi}$
1  $Cands \leftarrow \varnothing$;
2  for $\{v_1, \ldots, v_n\} \in paths(DT_P)$ do
3      for $i = 1, \ldots n$ do
4          $\ell_i \leftarrow \ell$;
5          for $j = (i-1), \ldots, 0$ do
6              $\ell_j \leftarrow$ BwPropagate($P, \ell_{j+1}, p(v_j), p(v_{j+1}), k$);
7          for $j = (i+1), \ldots, n$ do
8              $\ell_j \leftarrow$ FwPropagate($P, \ell_{j-1}, p(v_{j-1}), p(v_j), k$);
9          $Cands \leftarrow Cands \cup$
             $\{\lambda Vars\,.\,(p(v_m)(Vars) \implies \ell_m(Vars))\}_{m=0}^{n}$;
10 return $Cands$;
```

terminates (line 2). Otherwise, it picks a candidate ℓ that failed initiation (1) and/or consecution (2), then searches for a suitable phase among vertices of DT_P that can weaken ℓ (line 4). The weakening and new candidate generation is performed in Algorithm 6. Lastly, a set of new phase candidates is unified with the set of candidates to be checked by Houdini in the next recursive call of Algorithm 5.

A phase lemma has the form of implication, and it is guessed in Algorithm 6 (line 9) by composing a *guard*, as the left-hand side of the implication, and a formula ℓ, taken either from the initial candidate set or derived from it as the right-hand side. Algorithm 6 is iterative in nature: it generates sequences of phase candidates, following the actual paths of DT_P. That is, if a phase candidate is created for the initial candidate ℓ and an i^{th} phase of some path π, then the algorithm seeks to propagate ℓ backward to the $(i-1)^{\text{th}}$ phase of π, forward to $(i+1)^{\text{th}}$ phase, and if successful, even further.

Example 5.1. Recall Example 3.6 and failed candidate $z = 0$ for Example 3.4. Following the rightmost path of DT_P in Fig. 3, the algorithm guesses the first conjunct of invariant in Fig. 5 and proceeds to the next two phases.

Soundness of our algorithm is immediate: since based on Houdini, it only terminates with SAFE when all the conditions of Def. 3.3 are fulfilled by an external SMT solver.

Theorem 5.2 (Termination). *Assuming termination of Houdini and of the SMT solver, then Algorithm 5 will terminate.*

Given an initial set of candidate invariants, N, and a formula Tr, there can only be a finite number of phase guards, M, produced by quantifier elimination. These two pieces are the input to Algorithm 5. On each iteration a candidate, ℓ, is picked and sent to Algorithm 6 to attempt to connect it with a phase, line 9 of Algorithm 6. The loop in Algorithm 6 will terminate when all phases of the Tr have been explored (i.e., all paths in DT_P have been explored, of which there can be only a finite many), or when a candidate has been matched with its appropriate phase.

There are two conditions for Algorithm 5 to terminate, both of which are guaranteed. Either Houdini returns with a result SAFE (line 2), or the algorithm runs out of candidates to check and returns UNKNOWN (line 3). The algorithm considers at most a polynomial number of distinct candidates, meaning if no SAFE invariant is found, the result UNKNOWN is returned after the set of candidates is exhausted.

5.2 Optimizations

For readability purposes, Algorithm 6 is presented in an oversimplified form, so a direct implementation of which could be inefficient. Specifically, two outer nested loops attempt to match candidate ℓ with every possible phase (often, several times). In practice, there are many optimization opportunities (which we have in the implementation) that check the consistency of ℓ and each phase $p(v_i)$ before creating candidates. In particular, before proceeding to line 4, we require the following.

- For the head v_1 of every path, we require that $Init(Vars) \wedge p(v_1)(Vars) \implies \ell(Vars)$ is valid since otherwise the initiation condition (1) would not hold.
- Similarly, for every v_i, we require that $p(v_i)(Vars) \wedge \ell(Vars) \wedge Tr(Vars, Vars') \wedge p(v_{i+1})(Vars')$ is satisfiable (we do not use a stronger constraint involving the validity here since we may not know the precise reachability information at an arbitrary phase).

Additionally:

- Every edge of DT_P needs to be processed only once (for some path), and should be skipped in the other paths.
- Lemma propagation may be ineffective, i.e., depending on the strategy (to be presented in Sect. 6), it may return *true* or *false* candidates. The loops at lines 6 and 8 break in this case, and the algorithm proceeds to a new phase or candidate.

In practice, it is rarely the case that many candidates require all iterations of triple-nested loops in Algorithm 6. Recall, that input candidates to this algorithm are failed by Houdini, so our expectation then is that they will work only on a subset of phases/paths. Thus, in practice, the aforementioned optimizations are often sufficient to prevent the algorithm from diverging.

6 SYNTHESIS OF PHASE LEMMAS

In this section, we describe our lemma synthesis strategies for Algorithm 6 that are specifically tailored to a more semantically-aware search, based respectively on candidate propagation and data learning. These two approaches are used in Algorithm 6 at

$$
\begin{aligned}
x = 0 \land y = 767976 \land z = 0 &\implies Inv(x, y, z)\\
Inv(x, y, z) \land x' = x + 1 \land y' = y - 1 \land &\\
z' = ite((x - y) \ \mathtt{mod}\ 3 = 1, z + 3, z) &\implies Inv(x', y', z')\\
Inv(x, y, z) \land x \geq 280275 \land \neg(z \geq 280275) &\implies false
\end{aligned}
$$

Figure 6: Transition system encoding.

the call Fw/BwPropagate. The difference between these can be treated as a demarcation between a more-general instantiation of our algorithm (Sect. 6.1) and the one specific to arithmetic theories (Sect. 6.2).

6.1 Phase Propagation using Quantifier Elimination

Intuitively, once a candidate invariant and its phase guard is created, we can try to propagate it to the next phase, i.e., to a loop iteration where another phase guard holds, based on our DT_P. For any background theory that admits Quantifier Elimination (QE), Algorithm 6 can propagate candidates using Def. 6.1.

Definition 6.1. Given a transition relation $Tr(Vars, Vars')$, and a phase candidate $g(Vars) \implies \ell(Vars)$, and a next phase guard g_{next}, the FwPropagate method computes a candidate as follows:

$$
\begin{aligned}
&\mathrm{QE}(\exists Vars \,.\, Tr(Vars, Vars') \land g(Vars) \land\\
&\ell(Vars) \land g_{next}(Vars'))[Vars/Vars']
\end{aligned}
$$

The BwPropagate method computes a candidate as follows:

$$\mathrm{QE}(\forall Vars' \,.\, Tr(Vars, Vars') \land g_{next}(Vars) \land g(Vars') \implies \ell(Vars'))$$

Example 6.2. Fig. 6 gives a program with three phases, and its inductive invariant is as follows.

$$
\begin{aligned}
Inv \mapsto \lambda x, y, z \,.\, x + y = 767976 \land &\\
(x - y) \ \mathtt{mod}\ 3 = 0 &\implies x = z \land\\
(x - y) \ \mathtt{mod}\ 3 = 1 &\implies x - z = 2 \land\\
(x - y) \ \mathtt{mod}\ 3 = 2 &\implies x - z = 1
\end{aligned}
$$

Forward reasoning begins with an analysis of the initial state: $x+y = 767976$ and $x = z$ directly follow from the initial assignments to the variables. While the former is a global invariant in contrast, the latter needs the phase guard $(x - y) \ \mathtt{mod}\ 3 = 0$. In order to propagate this candidate forward, we note that under this phase, x grows by one and z does not change, thus if x was equal to z then $x' - z' = 1$, giving us the next candidate $x - z = 1$.

In practice, propagated candidates are often too strong because they describe exactly what happens in the next step after switching to a new phase. In many other cases, helper invariants need to be discovered prior to quantifier elimination, otherwise the synthesis procedure does not have enough information about the switching state. This motivates us to design alternative strategies for synthesizing phase lemmas in certain theories.

6.2 Fast-Forwarding to Data Candidates

Numeric theories enjoy well-known approaches like [60] to data-driven invariant generation. Gathering data for the early phases of an execution is trivial since the unrolling begins from the *Init* state, recall Sect. 3.3. A naive approach would unroll the program while the first phase guard holds. The resulting matrix would produce a candidate lemma for the phase, however, this is not scalable due to the taxing nature of large program unrollings. To solve this problem, we produce useful data in a more economical way with the notion of "fast-forwarding".[4]

Algorithm 7: FwPropagate($P, Inv(Vars), g, g_{next}, k$): data gathering at arbitrary point.

```
Input: $P = \langle Vars \cup Vars', Init, Tr \rangle$: program;
       $Inv$: invariant; $g$: phase-guard; $g_{next}$: the next
       phase-guard; $k$: unrolling bound
Output: $DM$: data matrix to be used for data learning
1  $unrl \leftarrow g(Vars) \land Inv(Vars) \land g_{next}(Vars')$;
2  for $c \in [1, k]$ do
3      $unrl \leftarrow unrl \land Tr(Vars^{(c)}, Vars^{(c+1)})$;
4  let $\mathcal{M}$ be s.t. $\mathcal{M} \models unrl$;
5  if $\mathcal{M} = \varnothing$ then
6      return $\varnothing$;
7  for $c \in [1, k]$ do
8      if $\mathcal{M} \models g_{next}(Vars^{(c)})$ then
9          $DM \leftarrow$ addRow($DM, \mathcal{M}(Vars^{(c)})$);
10 candFromGaussJordan($DM$);
11 return;
```

To start, a trace is produced as in Algorithm 7 (line 3) of some length k [5]. Then the supporting lemmas [6], a phase guard, and the next phase guard are added (line 1). By providing a phase guard ($g(Vars)$ in (4)) we are requiring the unrolling to begin at a particular point in the execution. Importantly, we also provide $Inv(Vars)$, all lemmas learned so far, and the phase lemmas associated with g.

$$
\begin{aligned}
&g(Vars) \land Inv(Vars) \land Tr(Vars, Vars') \land g_{next}(Vars') \land\\
&Tr(Vars', Vars'') \land \ldots \land Tr(Vars^{(k-1)}, Vars^{(k)})
\end{aligned} \tag{4}
$$

An invariant for the program in the example from Fig.7 requires a supporting lemma to discover, and must be provided to the solver before an unrolling begins (line 1 of Algorithm 7). Otherwise the invalid matrix **No Guard or Lemmas** in Fig. 8 may be produced.

If the solver is provided with the lemma $x \geq y$ (line 1), then it can produce valid information to create the other two matrices in Fig. 8, one for each phase. Each reveals a phase lemma that together with their phase guard verifies the program.

The **Even phase** matrix has a basis vector of $\left(\begin{smallmatrix}-1\\2\end{smallmatrix}\right)$ which yields the phase lemma $x = 2y$. Applying the next guard, $x \ \mathtt{mod}\ 2 = 1$ yields the phase lemma $x = 2y - 1$. The algorithm then terminates with the invariant:

$$
\begin{aligned}
Inv \mapsto \lambda x, y \,.\, x \ \mathtt{mod}\ 2 = 0 &\implies x = 2y \land\\
x \ \mathtt{mod}\ 2 = 1 &\implies x = 2y - 1
\end{aligned}
$$

[4] A "fast-backwarding" concept is defined similarly and skipped in the interest of space.
[5] Our implementation uses a value of 10 for k. This is done to keep unrollings small for performance reasons. However values of up to 30 have been tried with no improvement to the outcomes.
[6] Supporting lemmas are the previously learned lemmas from earlier iterations.

$$
\begin{aligned}
x = 0 \land y = 0 &\implies Inv(x, y) \\
Inv(x, y) \land x' = x + 1 \land & \\
y' = ite(x \bmod 2 = 0, y + 1, y) &\implies Inv(x', y') \\
Inv(x, y) \land x = 20000 \land \neg(y = 10000) &\implies false
\end{aligned}
$$

Figure 7: Transition system with alternating phases.

No Grd or Lms

x	y
-1	1
1	2
3	3

Even phase

x	y
2	1
4	2
6	3

Odd phase

x	y
1	1
3	2
5	3

Figure 8: Matrices produced for each phase of Fig. 7.

The importance of our "fast-forwarding" technique lies in its ability to synthesize difficult to infer lemmas, and associate them with their phase. The expression $x = 2y - 1$ is such an example, since on its own it is not easy to discover, and then to couple it with the correct phase guard is a difficult task. Fast-forwarding tackles those two challenges with the data learning approach connected to a particular phase.

7 IMPLEMENTATION

We have implemented our algorithm, called ImplCheck, in the latest version of the FreqHorn tool [18]. FreqHorn follows the SyGuS paradigm to guess-and-check for invariants derived from syntax and data. We leverage this design to start the ImplCheck algorithm. The decision tree containing MBPs is built in the early stages of an execution, and only once in an execution.

Candidates that fail the initial inductiveness check (2) move into Algorithm 5, where we identify the phase that the candidate belongs to in Algorithm 6. Algorithm 6 takes MBPs from the decision tree to test as phase-guards, described in Sect. 4. If a guard is successfully found, the guard and candidate are made into a phase-invariant in the form of implication.

New candidates are also discovered by our data learning technique from Sect. 6.2. We use phase guards and learned lemmas to create an unrolling and then obtain a data matrix from the unrolling. With "fast-forwarding" we are able to probe a particular phase of the program and connect lemmas discovered by the data learner with the phase guard as a new phase invariant.

Additionally, we leverage the *candidate propagation* feature of FreqHorn but make it more phase-aware (recall Sect. 6.1), which allows for lifting successful candidates to the next phase.

8 EVALUATION

We are interested in answering the following research questions:

- **RQ1**: How effective is ImplCheck in solving multi-phase benchmarks compared to state-of-the-art CHC solvers?
- **RQ2**: How crucial are data learning via fast-forwarding and MBP strengthening to solving multi-phase benchmarks?
- **RQ3**: How does ImplCheck perform on a large set of well studied benchmarks?

Throughout this section we discuss these questions in the context of our experimental results. The results are summarized in Tab. 1.

Experimental Results. ImplCheck has been compared against CHC solvers FreqHorn [18], PCSat [58], IC3IA [13], GSpacer [38], Spacer [37], Hoice [10], Eldarica [32], and Golem [8], and SyGuS solvers LoopInvGen [52], and CVC5 [54]. Since the input to all of these tools adheres to the SMT-LIB2 format (or the sister SyGuS-format), we do not compare to other software verification tools that would require additional symbolic execution/encoding of programs to symbolic constraints, making the experimental comparison less fair.

RQ1. We analyzed the performance of ImplCheck on 54 safe multi-phase benchmarks previously studied by Golem [8]. These programs are over integers and have single loops with a variety of phase structures.

With a 5 minute timeout, ImplCheck can solve 44 of the 54, the largest number of all the tools in the comparison. FreqHorn can solve 28. Eldarica performed the best out of the comparison tools, solving 25 of the multi-phase benchmarks. Eldarica's ability to solve these likely comes from its use of disjunctive interpolation [56]. GSpacer solved 19, Hoice, and Golem each solved 17. PCSat and IC3IA each solved 15 respectively, while CVC5 and Spacer solved 11 and 10. Finally, LoopInvGen solved 9. These results are displayed in Table 1.

There are 10 benchmarks that are only solved by ImplCheck, and one such example for the other tools, which is uniquely solved by Golem. These results show that ImplCheck is capable of verifying a variety of phase structures.

RQ2. We compared the performance of ImplCheck in various configurations, isolating the subsystems of the algorithm to highlight their impact. A summary of the results is in Table 2, which displays the lemma synthesis method, the direction of propagation, the number solved, the average and the median time to solve.

The subsystems are described in Sect. 4 - 6. In summary, PhaseData generates lemmas using the *fast-forwarding* data learning technique, PhaseProp propagates lemmas across phases, and StrenMBP strengthens the guards in the DT. FwdPropagate and BwdPropagate determine the direction of traversal through the program. By default both forward and backward propagation are enabled, but we explicitly disable one to parse out their impact.

Data learning with fast-forwarding, PhaseData, turns out to be the most impactful subsystem on its own. This feature, with forward propagation, solves 40 benchmarks. PhaseProp with backward propagation solves a similar number of benchmarks as forward, and proves to be a useful tactic for some cases, with two examples solved by Bwd that are not solved with Fwd.

The combination of the three options, PhaseData, PhaseProp, and StrenMBP, along with FwdPropagate solves the highest number of benchmarks at 41. There is a noticeable time cost across the tests however, with the median solve time around two seconds instead of below one second. PhaseData is able to solve many more of the examples below one second than the three options together can.

It is worth noting that even the worst configuration of ImplCheck, PhaseData Bwd, still outperforms all of the tools used for comparison.

Table 1: Timings in seconds (ϵ stands for a runtime less than a second; ∞ stands for not solved).

Benchmark	ImplCheck	FreqHorn	GSpacer	Spacer	IC3IA	LoopInvGen	Eldarica	CVC5	Hoice	PCSat	Golem
s_split_01	ϵ	1.88	4.38	∞	∞	∞	∞	∞	∞	∞	∞
s_split_02	ϵ	6.91	ϵ	62.10	∞	∞	∞	∞	ϵ	∞	188.31
s_split_03	1.63	73.55	∞	∞	∞	∞	1.06	∞	ϵ	14.39	∞
s_split_04	2.47	1.92	ϵ	∞	∞	∞	3.48	∞	5.29	∞	∞
s_split_05	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	2.40	ϵ	1.42	ϵ
s_split_06	ϵ	∞	2.35	∞	∞	∞	3.63	∞	∞	14.54	∞
s_split_07	ϵ	ϵ	ϵ	∞	ϵ	∞	1.11	20.50	ϵ	1.79	∞
s_split_08	2.83	2.83	∞	∞	∞	∞	∞	∞	∞	1.35	∞
s_split_09	ϵ	3.89	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_10	ϵ	10.84	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_11	ϵ	37.60	∞	∞	∞	∞	∞	∞	ϵ	∞	∞
s_split_12	ϵ	18.97	∞	∞	∞	∞	1.22	∞	∞	∞	∞
s_split_13	∞	∞	∞	∞	ϵ	ϵ	ϵ	10.37	ϵ	1.74	ϵ
s_split_14	∞	∞	∞	∞	∞	∞	26.38	∞	∞	∞	18.60
s_split_15	1.41	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_16	10.93	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_17	1.07	18.15	∞	∞	ϵ	∞	7.29	∞	∞	2.62	3.26
s_split_18	∞	∞	ϵ	∞	∞	∞	2.75	∞	ϵ	31.39	ϵ
s_split_19	ϵ	∞	ϵ	ϵ	ϵ	21.98	ϵ	∞	∞	∞	ϵ
s_split_20	ϵ	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_21	ϵ	ϵ	∞	53.81	21.31	∞	9.88	∞	ϵ	∞	ϵ
s_split_22	2.62	73.15	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_23	ϵ	ϵ	ϵ	∞	ϵ	1.89	1.52	ϵ	ϵ	1.51	ϵ
s_split_24	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	189.53
s_split_25	∞	∞	∞	∞	∞	∞	9.19	ϵ	∞	∞	ϵ
s_split_26	ϵ	ϵ	40.40	∞	ϵ	1.09	∞	ϵ	∞	1.24	∞
s_split_27	1.96	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_28	ϵ	54.00	∞	155.19	66.03	34.30	1.30	286.28	∞	∞	∞
s_split_29	108.69	∞	21.88	∞	∞	∞	32.18	∞	∞	∞	∞
s_split_30	∞	∞	∞	∞	3.17	∞	∞	∞	∞	∞	ϵ
s_split_31	4.72	4.72	∞	∞	∞	∞	1.29	∞	∞	∞	∞
s_split_32	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_33	ϵ	∞	∞	∞	∞	2.15	∞	∞	∞	∞	∞
s_split_34	1.99	∞	ϵ	126.86	253.89	3.94	5.24	∞	4.51	264.58	∞
s_split_35	∞	∞	ϵ	∞	∞	∞	1.55	∞	∞	∞	∞
s_split_36	ϵ	ϵ	ϵ	∞	ϵ	ϵ	1.08	ϵ	ϵ	1.24	ϵ
s_split_37	2.83	∞	ϵ	ϵ	ϵ	∞	13.94	∞	107.56	∞	ϵ
s_split_38	92.12	6.90	ϵ	∞	ϵ	∞	1.09	∞	ϵ	1.86	ϵ
s_split_39	ϵ	ϵ	8.48	44.05	2.08	∞	2.05	50.78	ϵ	1.37	ϵ
s_split_40	∞	∞	∞	153.13	∞	∞	∞	∞	∞	∞	4.44
s_split_41	ϵ	139.37	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_42	ϵ	∞	ϵ	∞	∞	∞	3.61	11.33	ϵ	1.56	∞
s_split_43	3.15	129.51	∞	∞	∞	∞	∞	∞	ϵ	∞	∞
s_split_44	1.22	104.14	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_45	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_46	ϵ	255.42	276.28	30.23	∞	∞	ϵ	33.33	∞	∞	∞
s_split_47	1.65	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_48	ϵ	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_49	ϵ	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_50	3.64	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_51	1.22	70.94	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_52	ϵ	88.14	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_53	3.24	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s_split_54	ϵ	58.05	∞	∞	∞	∞	∞	∞	∞	∞	∞
Total Solved	44	28	19	10	15	9	25	11	17	15	17
Uniquely Solved	10	0	0	0	0	0	0	0	0	0	1
Avg Time/Solved	6.18	41.48	18.78	62.58	23.13	7.45	5.38	37.77	7.07	22.84	23.80
Median Time/Solved	ϵ	8.88	ϵ	48.93	ϵ	1.89	1.55	10.37	ϵ	1.74	ϵ

Table 2: Summary of ImplCheck subsystems.

Benchmark	PhaseData BwdPropagate	PhaseData FwdPropagate	PhaseProp BwdPropagate	PhaseProp FwdPropagate	PhaseData + StrenMBP BwdPropagate	PhaseData + StrenMBP FwdPropagate	PhaseProp + StrenMBP BwdPropagate	PhaseProp + StrenMBP FwdPropagate	PhaseData + PhaseProp BwdPropagate	PhaseData + PhaseProp FwdPropagate	PhaseData + PhaseProp + StrenMBP BwdPropagate	PhaseData + PhaseProp + StrenMBP FwdPropagate
Total Solved	29	40	31	33	30	39	31	34	29	40	32	41
Avg Time/Solved	24.15	18.72	17.81	21.34	17.03	3.33	20.49	28.12	15.22	7.40	28.92	4.30
Median Time/Solved	ϵ	ϵ	ϵ	1.07	2.30	ϵ	1.03	1.06	1.23	1.72	3.06	2.06

Table 3: Summary of results over general benchmarks.

Benchmark	ImplCheck	FreqHorn	Eldarica	GSpacer	Golem	IC3IA
Total Solved	244	237	237	224	149	206
Avg Time/Solved	11.19	6.71	4.43	2.93	5.28	5.43
Median Time/Solved	3.43	1.93	1.06	ϵ	ϵ	ϵ

RQ3. We compared the performance of ImplCheck on a set of 277 well studied benchmarks taken from previous literature [16–19, 25]. This set of benchmarks covers both single loop and multi-loop programs, with the multi-loop programs having either nested loops or consecutive loops.

What we aim to show is that the overhead from ImplCheck is not prohibitively large on more general cases. The results in Table 3 show that ImplCheck is able to verify more benchmarks than the other tools, and the average overhead for examples solved by both ImplCheck and FreqHorn is only 4.5 seconds. ImplCheck solves 244 of the 277 and FreqHorn solves 237. This result is encouraging because it shows that ImplCheck is not pinned to solving one program type (ie. multi-phase) and adds only a small overhead to these examples.

9 THREATS TO VALIDITY

Our approach has been built under the consideration of LIA and the assumption that the underlying theory admits quantifier elimination for MBP generation. To extend this approach to other numeric theories we would need to consider how to generate MBPs since they are crucial to our phase finding algorithm. For Boolean programs, since our fast-forward algorithm relies on data learning with integers, we could instead rely on a QE approach to solve those problems.

ImplCheck is a CHC Solver and takes the output of a verification front end, like SeaHorn [27]. We rely on the correctness of the translation from the source code for the input to our algorithm. If the translation results in multi-phase loops, we believe our algorithm would have success verifying those cases.

Our analysis of program phases, and the construction of our DT_P, in a worst case scenario could be exponentially expensive. However, in practice this analysis is completed in less than a second for both multi-phase examples and the benchmarks in the large set, including MBP strengthening.

10 RELATED WORK

Inductive Invariant Synthesis. There are many automated software verification approaches that discover proofs in the form of an inductive invariant [1, 4, 9, 16, 23, 24, 26, 29–31, 34, 36, 37, 39, 41, 44]. There are many barriers to this problem, mainly due to its undecidability, requiring us to continue to look into this problem and consistently enlarge the applicability of the tools. While many methods have been successfully used for invariant synthesis, the vast majority do not reliably solve the case of multi-phased loops.

SyGuS [2] is also used for generating safe inductive invariants with the help of either a user provided grammar [52, 54], or an automatically generated one [18]. There are several positives to this approach, and ImplCheck takes advantage of enumeration early in the execution to find an initial set of candidates. Providing a detailed, fine-grain, grammar could in theory allow for the verification of multi-phase programs with previous SyGuS approaches. However, the grammar would need to include details about the construction of implications and the need for supporting lemmas.

Multi-phase Loops. Verification of loops with phases requires disjunctive invariants, and has been the subject of several works [5, 56, 64]. The studies to synthesize disjunctive invariants appears in several fields, suggesting this is an important problem to solve. Work on this problem has either relied on explicitly splitting a loop of consecutive phases [59], needed a user-provided grammar and interaction [22], or used abstraction techniques which "naturally" include disjunctions [64]. Other work on disjunctive invariant synthesis were approached by Abstract Interpretation [46, 57], Gröbner basis computation [33], and SyGuS [53]. An approach to loop summarization [61] reasons about program phases using "lazy" QE, but does not connect them with supporting invariants.

None of these offer as high a degree of flexibility as our approach: our fast-forwarding technique is effective in finding non-trivial invariants that are hard for techniques working in rigid domains, and they may require supporting invariants to complete verification. Our approach is also agnostic to the syntactic loop structure of a program and aims at *recovering* it from the symbolic encoding (that e.g., could flatten nested loops).

Data-driven Approaches. Previous work such as [20, 43, 47, 51, 60] use data derived from a given program to aid the inference of candidate invariants. At the core of these approaches is a *guess-and-check* structure, where the *guess* phase uses data collected from program traces and the *check* phase gathers data from counter-examples. Invariants in the form of both inequality and equality can be synthesized from traces using techniques from [48, 49]. Machine learning techniques for verification [25, 63] also use counter-example-derived data. It is our extension of the data-driven approach, with our fast-forwarding technique to produce a program trace at an arbitrary point in a programs execution (typically around the points of a phase change), that sets our work apart.

11 CONCLUSION

We have presented a novel approach to synthesize safe inductive invariants by *implicit splitting* of program phases, and we have augmented the FreqHorn algorithm to use our new technique, called ImplCheck. Our approach uses Model Based Projections to discover phase guards, stored in a decision tree, and it uses a data learning technique to discover phase lemmas. Key pieces of this approach are the ability to accurately fast-forward an unrolling to generate meaningful data and generate these two ingredients. We have demonstrated the usefulness of our approach by successfully solving a number of challenging verification tasks.

Motivation for future improvements to our approach include investigating the benchmarks solved by other tools but not ImplCheck and improving the efficiency of our algorithm. Support for arrays and algebraic data types is also a goal for future development.

Artifact Available. A virtual machine is available to reproduce the reported results [55].

Acknowledgements. This work was partially supported by the National Science Foundation grant 2106949.

REFERENCES

[1] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012. From Under-Approximations to Over-Approximations and Back. In *TACAS (LNCS, Vol. 7214)*. Springer, 157–172.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *FMCAD*. IEEE, 1–17.

[3] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A constraint-based approach to solving games on infinite graphs. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 221–234. https://doi.org/10.1145/2535838.2535860

[4] Dirk Beyer, Matthias Dangl, and Philipp Wendler. 2015. Boosting k-Induction with Continuously-Refined Invariants. In *CAV, Part I (LNCS, Vol. 9206)*. 622–640.

[5] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. 2007. Path invariants. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 300–309. https://doi.org/10.1145/1250734.1250769

[6] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. 2003. Bounded Model Checking. *Advances in Computers* 58 (2003), 118–149.

[7] Nikolaj Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. In *LPAR (short papers) (EPiC Series in Computing, Vol. 35)*. EasyChair, 15–27.

[8] Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. 2022. Transition Power Abstractions for Deep Counterexample Detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer Berlin Heidelberg.

[9] Aaron R. Bradley. 2012. Understanding IC3. In *SAT (LNCS, Vol. 7317)*. Springer, 1–14.

[10] Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. 2018. HoIce: An ICE-Based Non-linear Horn Clause Solver. In *APLAS (LNCS, Vol. 11275)*. Springer, 146–156.

[11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *PLDI*. ACM, 415–426.

[12] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *TACAS (LNCS, Vol. 7795)*. Springer, 47–61.

[13] Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. 2016. Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations. In *CAV, Part I (LNCS, Vol. 9779)*. Springer, 271–291.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340.

[15] Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *CAV (LNCS, Vol. 8044)*. 684–689.

[16] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *OOPSLA*. ACM, 443–456.

[17] Grigory Fedyukovich and Rastislav Bodík. 2018. Accelerating Syntax-Guided Invariant Synthesis. In *TACAS, Part I (LNCS, Vol. 10805)*. Springer, 251–269.

[18] Grigory Fedyukovich, Samuel Kaufman, and Rastislav Bodík. 2017. Sampling Invariants from Frequency Distributions. In *FMCAD*. IEEE, 100–107.

[19] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2018. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*. IEEE, 170–178.

[20] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2019. Quantified Invariants via Syntax-Guided Synthesis. In *CAV, Part I (LNCS, Vol. 11561)*. Springer, 259–277.

[21] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. 2018. Syntax-Guided Termination Analysis. In *CAV, Part I (LNCS, Vol. 10981)*. Springer, 124–143.

[22] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. 2019. Inferring Inductive Invariants from Phase Structures. In *CAV, Part II (LNCS, Vol. 11562)*. Springer, 405–425.

[23] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini: an Annotation Assistant for ESC/Java. In *FME (LNCS, Vol. 2021)*. Springer, 500–517.

[24] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *CAV (LNCS, Vol. 8559)*. Springer, 69–87.

[25] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *POPL*. ACM, 499–512.

[26] Sumit Gulwani and Nebojsa Jojic. 2007. Program verification as probabilistic inference. In *POPL*. ACM, 277–289.

[27] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV (LNCS, Vol. 9206)*. Springer, 343–361.

[28] Arie Gurfinkel and Jorge A Navas. 2019. Automatic Program Verification with SEAHORN. In *Engineering Secure and Dependable Software Systems*. IOS Press, 83–111.

[29] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested interpolants. In *POPL*. ACM, 471–482.

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from proofs. In *POPL*. ACM, 232–244.

[31] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. 2012. A Verification Toolkit for Numerical Transition Systems - Tool Paper. In *FM (LNCS, Vol. 7436)*. Springer, 247–251.

[32] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *FMCAD*. IEEE, 158–164.

[33] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2018. Invariant Generation for Multi-Path Loops with Polynomial Assignments. In *VMCAI (LNCS, Vol. 10747)*. Springer, 226–246.

[34] Dejan Jovanovic and Bruno Dutertre. 2016. Property-directed k-induction. In *FMCAD*. IEEE, 85–92.

[35] Andreas Katis, Grigory Fedyukovich, Huajun Guo, Andrew Gacek, John Backes, Arie Gurfinkel, and Michael W. Whalen. 2018. Validity-Guided Synthesis of Reactive Systems from Assume-Guarantee Contracts. In *TACAS, Part II (LNCS, Vol. 10806)*. Springer, 176–193.

[36] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *PLDI*. ACM, 248–262.

[37] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *CAV (LNCS, Vol. 8559)*. 17–34.

[38] Hari Govind Vediramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. 2020. Global Guidance for Local Generalization in Model Checking. In *CAV (LNCS, Vol. 12225)*. Springer, 101–125.

[39] Hari Govind Vediramana Krishnan, Grigory Fedyukovich, and Arie Gurfinkel. 2020. Word Level Property Directed Reachability. In *ICCAD*. IEEE, 1–9.

[40] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and non-termination specification inference. In *PLDI*. ACM, 489–498.

[41] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *PLDI*. ACM, 788–801.

[42] Jongeun Lee, Seongseok Seo, Hongsik Lee, and Hyeon Uk Sim. 2014. Flattening-based mapping of imperfect loop nests for CGRAs?. In *2014 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2014, Uttar Pradesh, India, October 12-17, 2014*, Radu Marculescu and Gabriela Nicolescu (Eds.). ACM, 9:1–9:10. https://doi.org/10.1145/2656075.2656085

[43] Hong Lu, Jiacheng Gui, Chengyi Wang, and Hao Huang. 2020. A Novel Data-Driven Approach for Generating Verified Loop Invariants. In *International Symposium on Theoretical Aspects of Software Engineering, TASE 2020, Hangzhou, China, December 11-13, 2020*, Toshiaki Aoki and Qin Li (Eds.). IEEE, 9–16. https://doi.org/10.1109/TASE49443.2020.00011

[44] Kenneth L. McMillan. 2014. Lazy Annotation Revisited. In *CAV (LNCS, Vol. 8559)*. Springer, 243–259.

[45] David Monniaux. 2010. Quantifier Elimination by Lazy Model Enumeration. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 585–599. https://doi.org/10.1007/978-3-642-14295-6_51

[46] David Monniaux and Martin Bodin. 2011. Modular Abstractions of Reactive Nodes Using Disjunctive Invariants. In *APLAS (LNCS, Vol. 7078)*. Springer, 19–33.

[47] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided approach to finding numerical invariants. In *ESEC/FSE*. ACM, 605–615.

[48] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 683–693. https://doi.org/10.1109/ICSE.2012.6227149

[49] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (2014), 30:1–30:30. https://doi.org/10.1145/2556782

[50] Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. 2019. Overfitting in Synthesis: Theory and Practice. In *CAV, Part I (LNCS, Vol. 11561)*. Springer, 315–334.

[51] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56.

[52] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *PLDI*. 42–56. https://doi.org/10.1145/2908080.2908099

[53] Sumanth Prabhu, Kumar Madhukar, and R Venkatesh. 2018. Efficiently learning safety proofs from appearance as well as behaviours. In *SAS (LNCS, Vol. 11002)*. Springer, 326–343.

[54] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *CAV, Part II (LNCS, Vol. 11562)*. Springer, 74–83.

[55] Daniel Riley and Grigory Fedyukovich. 2022. Artifact for Multi-Phase Invariant Synthesis. (Aug 2022). https://doi.org/10.5281/zenodo.7047061

[56] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2013. Disjunctive Interpolants for Horn-Clause Verification. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 347–363. https://doi.org/10.1007/978-3-642-39799-8_24

[57] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. 2006. Static Analysis in Disjunctive Numerical Domains. In *SAS (LNCS, Vol. 4134)*. Springer, 3–17.

[58] Yuki Satake, Hiroshi Unno, and Hinata Yanagi. 2020. Probabilistic Inference for Predicate Constraint Satisfaction. In *AAAI*. AAAI Press, 1644–1651. https://aaai.org/ojs/index.php/AAAI/article/view/5526

[59] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *CAV (LNCS, Vol. 6806)*. Springer, 703–719.

[60] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *ESOP (LNCS, Vol. 7792)*. Springer, 574–592.

[61] Jake Silverman and Zachary Kincaid. 2019. Loop Summarization with Rational Vector Addition Systems. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 97–115. https://doi.org/10.1007/978-3-030-25543-5_7

[62] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing Ranking Functions from Bits and Pieces. In *TACAS (LNCS, Vol. 9636)*. Springer, 54–70.

[63] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *PLDI*. ACM, 707–721.

[64] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 280–297. https://doi.org/10.1007/978-3-642-23702-7_22