Data-Driven Loop Invariant Inference with Automatic Feature Synthesis

(Competition Contribution)

Saswat Padhi
University of California, Los Angeles
Email: padhi@cs.ucla.edu

Todd Millstein
University of California, Los Angeles
Email: todd@cs.ucla.edu

***Abstract*—We present LoopInvGen, a tool for generating loop invariants that can provably guarantee correctness of a program with respect to a given specification. We extend the data-driven approach to inferring sufficient loop invariants from a collection of program states. In contrast to existing data-driven techniques, LoopInvGen is not restricted to a fixed set of *features* – atomic predicates that are composed together to build complex loop invariants. Instead, we start with no initial features, and use program synthesis techniques to grow the set on demand.**

Compared to existing static and dynamic techniques for loop invariant inference, not only does LoopInvGen enable a less onerous and more expressive approach, but is also significantly faster over the SyGuS-INV (2017) benchmarks.

I. Introduction

Formally proving the correctness of a program with respect to a given specification, can be largely automated when the appropriate *program invariants* are available. Yet, the problem of learning the adequate invariants in the first place, remains quite challenging. Traditional *static* approaches that reason over the program structure to deduce sufficient invariants, are often inapplicable to real-life cases simply because the program logic is far too complex to be analyzable. However, it is often the case that complex real-life programs have relatively simple invariants that certify their correctness relative to properties of practical interest. In such cases, *data-driven* approaches seem to perform well. These techniques learn a candidate invariant by examining program behavior (as opposed to structure), and then refine it till it is sufficiently strong.

We extend the data-driven paradigm for inferring sufficient loop invariants. Given some sets of "good" and "bad" program states, data-driven approaches learn a candidate invariant as a boolean combination of atomic predicates (called *features*) defined on states, such that it is satisfied by the good states and falsified by the bad ones. Prior techniques are restricted to using a fixed set, or a fixed template for features. For instance, a state-of-the-art technique, ICE-DT [1] requires the shape of constraints (such as octagonal) to be fixed apriori[1]. A fixed set of features not only limits the expressiveness, but predicting such a set, which would be adequate for learning a sufficiently strong invariant is also quite challenging [2].

We present LoopInvGen, a data-driven tool for inferring sufficient loop invariants, which starts with no initial features, and automatically learns features as necessary using *program synthesis* techniques. LoopInvGen is an optimized implementation of the general inference technique proposed in our recent work on data-driven precondition inference [2]. It reduces the problem of loop invariant inference to a series of precondition inference problems, and alternates between two phases to converge to a sufficient invariant: **(1)** *learning* a candidate invariant by solving the appropriate precondition inference problem, and **(2)** *checking* if the learned candidate is sufficient for proving correctness. If a candidate is insufficient, a *counterexample* is extracted from the checker, and is used to guide the learning phase towards the desired invariants.

Our technique is modular, and makes no assumptions on the specific program synthesizer used for feature synthesis, except that the language of the synthesizer must be compatible with the theorem prover employed for checking. The synthesizer utilized by LoopInvGen is currently restricted to expressions over the theory of *linear integer arithmetic* (LIA), which is the sole focus of the INV track of SyGuS-COMP 2018.

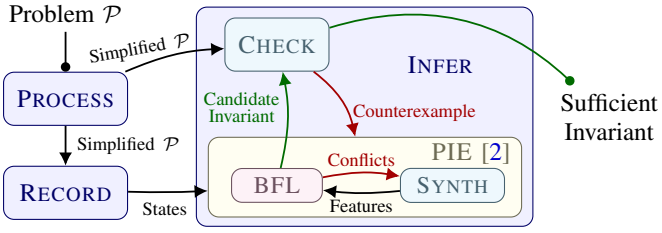

Fig. 1: The key components in LoopInvGen, and their interdependence.

II. Overview

Figure 1 shows a high-level schematic of LoopInvGen. It consists of three major components: **(1)** Process – performs some simplifications using static analysis, **(2)** Record – collects the data required to drive the inference, and **(3)** Infer – uses the PIE and Check subcomponents to learn candidate invariants, and verify that they satisfy the desired properties.

In the following subsections, we briefly describe each of these subcomponents, and illustrate them with the help of a running example. We consider a program, listed in Fig. 2, in which x is iteratively doubled starting from 1 till $(x \geqslant y)$, and y may be arbitrarily updated at each iteration. The goal

[1] ICE-DT also requires specialized learners for boolean formulas, which can utilize the *implication counterexamples*.

```
(set-logic LIA)

(synth-inv inv-f ((x Int) (y Int)
                  (z1 Int) (z2 Int) (z3 Int)))

(declare-primed-var x Int)
(declare-primed-var y Int)
(declare-primed-var z1 Int)
(declare-primed-var z2 Int)
(declare-primed-var z3 Int)

(define-fun pre-f ((x Int) (y Int)
                   (z1 Int) (z2 Int) (z3 Int)) Bool
  (= x 1))


(define-fun trans-f ((x Int) (y Int)
                     (z1 Int) (z2 Int) (z3 Int)
                     (x! Int) (y! Int)
                     (z1! Int) (z2! Int) (z3! Int)) Bool
  (and (< x y) (= x! (+ x x))))

(define-fun post-f ((x Int) (y Int)
                    (z1 Int) (z2 Int) (z3 Int)) Bool
  (or (not (>= x y)) (>= x 1)))

(inv-constraint inv-f pre-f trans-f post-f)

(check-synth)
```

Fig. 2: The `trex1_vars` benchmark from SyGuS-COMP 2016 (INV track).

is to verify that $(x \geqslant 1)$ always holds after the loop. The SyGuS-INV format [3] used in Fig. 2, allows encoding the semantics of the program along with a desired functional specification. For the remainder of the paper, we use the triplet $\langle P, T, Q, \Delta\rangle$to denote an arbitrary SyGuS-INV problem — P being the precondition, Q the postcondition, T the state transition relation, and Δ the remaining facts, if any.

A. PROCESS: *Simplification using Static Analysis*

This first component statically analyzes a given SyGuS-INV problem, and generates a simplified problem which is propagated to the subsequent components. Moreover, it also performs basic syntactic and semantic checks to ensure validity of the problem, and serializes it to a binary format that can be directly deserialized, eliminating the need to re-parse the specification within the subsequent components.

Currently, the PROCESS component only performs an *unused variable elimination* over a given SyGuS-INV problem $\langle P, T, Q, \Delta\rangle$. For this analysis, we define "use" of a variable v as its presence within either the specification (P or Q), or the state transition relation (v or v' in T), upon inlining all other relations from Δ. This analysis reduces the variables that we consider during invariant synthesis later – unused program variables should not affect the validity of the postcondition.

To eliminate unused variables, we first construct a call graph of all the relations, and perform 3 topological sorts over them rooted at P, Q and T. Then, starting with the leaf nodes in each sorted order, we label the "used" formal parameters V_R for each relation R, referring to the labels assigned to its callees' formal parameters, at each invocation point. Finally, we compute the set of all used variables as:

$$V = V_P \cup \{v \mid v \in V_T \vee v' \in V_T\} \cup V_Q$$

func RECORD($\langle P, T, Q, \Delta\rangle$: SyGuS$_{\text{INV}}$, V: `String[]`, n: `Int`)
Result: A collection of program states $\mathcal{Z}$: `State[]`.

```
1  Z ← {}
2  while true do
►      Start with a previously unseen model of the precondition.
3      m ← GetModel(Δ ∧ P(m) ∧ (⋀_{s ∈ Z} m ≠ s), V)
4      if m = None then break
5      Z ← Z ∪ RecordStatesFrom(m, n)
6      if |Z| = n then break
7  return Z
```

func RECORDSTATESFROM(m: `State`, k: `Int`)
Result: A sequence $\{m, m_1, m_2, \cdots, m_l\}$ of states, where $l \leqslant k$.

```
1  Z ← {m}
2  while |Z| < n do
►      Make a transition, i.e. execute a single iteration of the loop.
3      m ← GetModel(Δ ∧ T(m, m'), {v' | v ∈ V})
4      if m = None then break else Z ← Z ∪ {m}
5  return Z
```

Fig. 3: An outline of the RECORD component of LOOPINVGEN.

For our running example from Fig. 2, we would have:

$$V_P = \{x\} \;;\; V_T = \{x, y, x'\} \;;\; V_Q = \{x, y\} \qquad V = \{x, y\}$$

B. RECORD: *Collecting Reachable Program States*

This component collects a sample of the program states reachable at the two locations where a loop invariant must hold – **(1)** the beginning of each loop iteration, and **(2)** just after exiting the loop. To collect these states for programs encoded in the SyGuS-INV format [3], we use a constraint solver as an execution engine[2]. We present an outline of the RECORD algorithm in Fig. 3, which invokes a constraint solver within the GETMODEL procedure. The algorithm accepts a SyGuS-INV problem $\langle P, T, Q, \Delta\rangle$, the set V of variables to track, the desired number n of program states, and returns the set $\mathcal{Z}$ of the states of the variables in V.

In line 3, we start with an unseen model of the precondition, which is a state of the program at beginning of the first iteration. For instance, $(x \mapsto 1)$ is one such model for our running example from Fig. 2, with $V = \{x, y\}$. The GETMODEL function accepts a predicate, a list of variables, and returns a satisfying assignment for them. Note that this is not a *complete* state of the program since the variable y is unbound. In such cases, GETMODEL employs a pseudo-random number generator to extend the model to a complete program state, assigning arbitrary values to unconstrained variables. For our running example, such program states could, for instance, be $(x \mapsto 1 \wedge y \mapsto -3)$, or $(x = 1 \wedge y = 7)$ etc.

In lines 5 – 8, we execute several iterations of the loop body, and collect the program states at the loop head each time. In the SyGuS-INV encoding, executing a single iteration of the loop is equivalent to making a transition from the current state. In line 6, we solve for the next program state resulting from such a transition, and save it to $\mathcal{Z}$ in line 7. For our running example, the state $(x \mapsto 1 \wedge y \mapsto 7)$ will transition to $(x \mapsto 2)$,

[2] Our original technique [2] instrumented C/C++ programs, and collected program states during execution of the program.

```
func INFER(⟨P, T, Q, Δ⟩ : SyGuS_INV, Z : State[], Θ : Config)
Result: A sufficient loop invariant I : State → Bool.
► Start with the weakest invariant that satisfies ∀s : I(s) ⇒ Q(s).
1  I ← Q
► Iteratively strengthen I till it is inductive.
2  while true do
3      B ← {}
4      while true do
5          ρ ← PIE(Z, B, Θ)
6          c ← CHECK(∀s, t : ρ(s) ⇒ I(s) ∧ T(s, t) ⇒ I(t))
7          if c = None then break else B ← B ∪ {c}
8      I ← I ∧ ρ
►      Weaken I using counterexamples, if it is stronger than P.
9      c ← CHECK(∀s : P(s) ⇒ I(s))
10     if c ≠ None then
11         S ← RECORDSTATESFROM(c, Θ[NumStepsOnRestart])
12         return INFER(⟨P, T, Q, Δ⟩, Z ∪ S)
13     else if ρ = true then break
14 return I
```

Fig. 4: An outline of the INFER component of LOOPINVGEN.

that could be extended to $(x \mapsto 2 \wedge y \mapsto -2)$, for example. Note that no further transitions are possible from this state, since $2 \not< -2$ (implicit loop guard in the transition relation).

If we reach such a state from which no transitions are possible, and the set $\mathcal{Z}$ of collected program states contains less than the desired number n of states then, in line 3, we start with an *unseen* state (which is not already in the set $\mathcal{Z}$).

C. INFER: *Inference of Sufficiently Strong Loop Invariants*

This component uses the program states collected by RECORD to infer a loop invariant that is sufficient for proving correctness of a given SyGuS-INV problem $\langle P, T, Q, \Delta \rangle$. We outline our INFER algorithm in Fig. 4, which given a SyGuS-INV problem $\langle P, T, Q, \Delta \rangle$, a set $\mathcal{Z}$ of reachable states, and a set Θ of configuration parameters, returns an invariant $\mathcal{I}$.

A *sufficient* loop invariant $\mathcal{I}$ must satisfy three conditions:

- Weaker than precondition: $\forall s : P(s) \Rightarrow \mathcal{I}(s)$
- Inductive over loop body: $\forall s, t : \mathcal{I}(s) \wedge T(s, t) \Rightarrow \mathcal{I}(t)$
- Stronger than postcondition: $\forall s : \mathcal{I}(s) \Rightarrow Q(s)$

As shown in Fig. 1, INFER relies on an off-the-shelf theorem prover CHECK for verifying these conditions, and employs PIE [2] to refine candidate invariants. In line 1, it starts with the weakest possible candidate[3], $\mathcal{I} = Q$, and iteratively refines $\mathcal{I}$ till all of the above properties are satisfied. For instance, on our running example from Fig. 2, INFER starts with the initial candidate invariant $\mathcal{I}_0 = ((x < y) \vee (x \geqslant 1))$.

However, this candidate invariant is not inductive. The state $(x \mapsto 0 \wedge y \mapsto 1)$ satisfies $\mathcal{I}$, but it may transition to state $(x \mapsto 0 \wedge y \mapsto 0)$, which violates $\mathcal{I}$. In lines 2 – 13, INFER employs a *strengthening* loop (inspired by HOLA [4]), to ensure inductiveness of the candidate. At the i^{th} iteration, it learns a precondition ρ_i under which the candidate invariant is preserved after a single transition. For our running example, $\rho_1 = (x \geqslant 1)$, for instance, would ensure that our candidate invariant $\mathcal{I}_0 = ((x < y) \vee (x \geqslant 1))$ is preserved. In line 8, we strengthen the candidate invariant by conjoining it with the learned precondition. For our running example, the new candidate $\mathcal{I}_1 = \mathcal{I}_0 \wedge \rho_1 = (x \geqslant 1)$ is indeed inductive.

The reduction to a precondition inference problem allows us to leverage our prior work, PIE, on learning preconditions with automatic synthesis of appropriate features[4]. In line 5, PIE accepts a set $\mathcal{Z}$ of states which lead to satisfaction of a desired property, a set B of states which do not, the set Θ of configuration parameters (such as *conflict group size* [2]), and learns a *likely* precondition ρ for the desired property. Since the precondition is only a likely one, in line 6, INFER checks the likely precondition using CHECK for sufficiency, and provides counterexamples to PIE iteratively, in lines 4 – 7, till a provably sufficient precondition is learned.

Conjoining the current candidate invariant $\mathcal{I}$ with the precondition ρ, might however result in the next candidate $\mathcal{I} \wedge \rho$ being too strong, in particular, stronger than the precondition P. Therefore, in line 9, we use CHECK to verify that it is weaker than P. A counterexample in this case would indicate a state that is allowed by the precondition, but not covered by the candidate invariant. This could happen due to inadequate exploration of program states during the RECORD phase, for instance due to a complex transition relation. On finding such a counterexample c, we invoke RECORDSTATESFROM (from Fig. 3) to collect a few more (NumStepsOnRestart parameter in Θ) states starting from c, in line 11, to account for the unexplored program behavior. Finally, in line 12, we restart with the new set of available program states. Note that if no such counterexample is found and the current candidate *unconditionally* holds (i.e. $\rho =$ `true`), as is the case with the candidate $\mathcal{I} = (x \geqslant 1)$ for our running example, then our current candidate invariant is provably sufficient for guaranteeing correctness of $\langle P, T, Q, \Delta \rangle$.

III. IMPLEMENTATION

Our implementation of LOOPINVGEN is open source, and is available at https://github.com/SaswatPadhi/LoopInvGen. For its various components, LOOPINVGEN internally uses the following off-the-shelf algorithms or implementations:

- Both GETMODEL and CHECK are implemented using the Z3 [5] theorem prover. Our prior work used CVC4 [6] for reasoning over the theory of strings, which is beyond the scope of INV track of SyGuS-COMP 2018.
- PIE uses the ESCHER [7] program synthesizer as its SYNTH component. The language for synthesis has been shrunk to only allow expressions over LIA theory.
- The BFL component in PIE uses a standard *probably approximately correct* (PAC) algorithm that can learn arbitrary *conjunctive normal form* (CNF) formula, and is biased towards small formulas [8].

[3] Our original technique [2] used PIE to learn the initial candidate invariant $\mathcal{I}$ as one that satisfies $\{\mathcal{I}\}$ **skip** $\{Q\}$. We found this initial candidate to be too strong sometimes, requiring additional counterexamples to weaken it.

[4] PIE uses two off-the-shelf components: **(1)** a program synthesizer SYNTH to generate new features, and **(2)** a boolean function learner BFL to learn a composition of these features. The details are presented in our full paper [2].

Major Implementation Changes

Since SyGuS-COMP 2017:

- PROCESS– We now have a static analysis pass before the RECORD and INFER components (see Section II-A).
- Early Precondition Check – As opposed to finally checking if an inductive invariant is weaker than the precondition, we now check this property at each strengthening.
- AST Pruning – We have implemented a syntactic checking phase before ESCHER's semantic checks, that prunes redundant ASTs such as (`_ + x - x`) or (`1 * _`) etc.
- Better SyGuS-INV Support – We have added support for defining and invoking relations with arbitrary sorts, other than precondition, postcondition and transition relations.
- Beyond LIA Theory – We have implemented experimental support for theory of Non-Linear Integer Arithmetic (NLIA), which may be activated using the command: `(set-logic NLIA)`.

Since First Publication [2]:

- RECORD Coverage – The RECORD component has been significantly improved to better explore program states for non-deterministic programs. Along with a better selection of initial candidate invariant, this allowed us to start with only 512 program states instead of 6400.
- Parallel RECORD – Multiple (by default, 2) instances of RECORD with different seeds for PRNGs are run in parallel, and the program states are then merged.
- Z3 Scopes – LOOPINVGEN creates a single subprocess for Z3, and relies heavily on scopes to cache context information, and minimize the size of queries.
- Unsolvability Detection – LOOPINVGEN immediately terminates if $\exists s\colon P(s) \not\Rightarrow Q(s)$, i.e. the precondition does not imply the postcondition. It also keeps track of known program states, and terminates as soon as a state appears to be a negative example (w.r.t. the given specification).
- *Conflict Group Size* [2] – Overriding PIE's default size of 16, LOOPINVGEN uses 64.

IV. Conclusion

We have described LOOPINVGEN, which uses a data-driven approach to generate loop invariants that provably guarantee the correctness of an implementation with respect to a given specification. In contrast to existing techniques, LOOPINVGEN **(1)** is not restricted to any specific logical theory, and **(2)** starts with no initial features and learns them automatically on demand. In essence, LOOPINVGEN reduces the problem of loop invariant inference to a series of precondition inference problems, and solves them using PIE, which uses a form of program synthesis to learn features in a targeted manner.

Acknowledgment

Thanks to Rahul Sharma for his contributions to our joint prior work on PIE, and the initial prototype of LOOPINVGEN; Sumit Gulwani and Zachary Kincaid for access to the ESCHER program synthesis tool; and the organizers of SyGuS-COMP for making all the artifacts publicly available.

References

[1] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning Invariants using Decision Trees and Implication Counterexamples," in *POPL*, 2016.

[2] S. Padhi, R. Sharma, and T. D. Millstein, "Data-Driven Precondition Inference with Learned Features," in *PLDI*, 2016.

[3] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "SyGuS-Comp 2016: Results and Analysis," in *SYNT @ CAV*, 2016.

[4] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, "Inductive invariant generation via abductive inference," in *OOPSLA*, 2013.

[5] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, 2008.

[6] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, 2011.

[7] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive Program Synthesis," in *CAV*, 2013.

[8] M. Kearns and U. V. Vazirani, "An Introduction to Computational Learning Theory," 1994.