UCLA
UCLA Previously Published Works

Title
Data-Driven Loop Invariant Inference with Automatic Feature Synthesis.

Permalink
https://escholarship.org/uc/item/877943sm

Journal
CoRR, abs/1707.02029

Authors
Padhi, Saswat
Millstein, Todd D

Publication Date
2017

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Data-Driven Loop Invariant Inference with Automatic Feature Synthesis

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

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

arXiv:1707.02029v1 [cs.PL] 7 Jul 2017

***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.**

We compare with existing static and dynamic tools for loop invariant inference, and show that LoopInvGen enables a less onerous and more expressive form of inference.

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 *white-box* or *static* inference 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 many complex real-life programs have relatively simple invariants that certify their correctness relative to properties of practical interest. In such cases, *black-box* or *data-driven* inference techniques 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 were 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 grows the feature set 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.

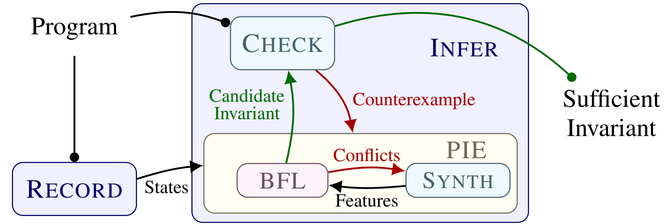

Fig. 1: The key components in LoopInvGen, and their interdependence.

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 2017.

II. Overview

Figure 1 displays a high-level schematic of LoopInvGen, as proposed in our prior work [2]. It consists of two major components: **(1)** Record, which collects the data required to drive the inference, and **(2)** Infer, which 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 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.

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

```
1  (set-logic LIA)

3  (synth-inv inv-f ((x Int) (y Int)))

5  (declare-primed-var x Int)
6  (declare-primed-var y Int)

8  (define-fun pre-f ((x Int) (y Int)) Bool (= x 1))

10 (define-fun trans-f ((x Int) (y Int)
11                      (x! Int) (y! Int)) Bool
12   (and (< x y) (= x! (+ x x))))

14 (define-fun post-f ((x Int) (y Int)) Bool
15   (or (not (>= x y)) (>= x 1)))

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

19 (check-synth)
```

Fig. 2: The `treax1.sl` benchmark from SyGuS-COMP 2016 (INV track).

A. RECORD: *Collecting 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 the specified precondition P, the transition function T, the desired number n of program states, and returns the set $\mathcal{Z}$ of states.

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. 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 $(x \mapsto 2)$, 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 function).

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 $\mathcal{Z}$).

[2] Our original technique [2] instrumented C/C++ programs to collect these program states during execution.

func RECORD(P: Pred⟨State⟩, T: Pred⟨State × State⟩, n: Int)
Result: A collection of program states $\mathcal{Z}$: State[].

1 $\mathcal{Z} \leftarrow \{\}$
2 **while** $|\mathcal{Z}| < n$ **do**
▶ Start with a *new* model of the precondition.
3 $x \leftarrow$ GETMODEL($P(x) \wedge (\bigwedge_{s \in \mathcal{Z}} x \neq s)$, x)
4 **if** $x =$ None **then break else** $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{x\}$
5 **repeat**
▶ Make a transition, i.e. execute a single iteration of the loop.
6   $x \leftarrow$ GETMODEL($T(x, x')$, x')
7   **if** $x =$ None **then break else** $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{x\}$
8 **until** $|\mathcal{Z}| < n$
9 **return** $\mathcal{Z}$

func INFER(P: Pred⟨State⟩, T: Pred⟨State × State⟩, Q: Pred⟨State⟩, $\mathcal{Z}$: State[])
Result: A sufficient loop invariant $\mathcal{I}$: Pred⟨State⟩.

▶ Start with the weakest invariant that satisfies $\forall s\colon \mathcal{I}(s) \Rightarrow Q(s)$.
1 $\mathcal{I} \leftarrow Q$
▶ Iteratively strengthen the invariant till it is inductive.
2 **while** CHECK($\forall p, q\colon \mathcal{I}(p) \wedge T(p, q) \Rightarrow \mathcal{I}(q)$) $\neq$ None **do**
3 $B \leftarrow \{\}$
4 **repeat**
5   $\Delta \leftarrow$ PIE($\mathcal{Z}$, B)
6   $c \leftarrow$ CHECK($\forall p, q\colon \Delta(p) \Rightarrow \mathcal{I}(p) \wedge T(p, q) \Rightarrow \mathcal{I}(q)$)
7   $B \leftarrow B \cup \{c\}$
8 **until** $c =$ None
9 $\mathcal{I} \leftarrow \mathcal{I} \wedge \Delta$
▶ Weaken the invariant using counterexamples, if it is stronger than P.
10 $C \leftarrow$ CHECK($\forall s\colon P(s) \Rightarrow \mathcal{I}(s)$)
11 **if** $C \neq$ None **then**
12 $\mathcal{S} \leftarrow$ RECORDSTATESFROM(C, k)
13 **return** INFER($P, Q, T, \mathcal{Z} \cup \mathcal{S}$)
14 **return** $\mathcal{I}$

Fig. 3: An outline of the algorithms employed by the two key components of LOOPINVGEN – RECORD and INFER.

B. INFER: *Inference of Sufficient Invariants*

This component uses program states collected by RECORD to infer a sufficient loop invariant for proving correctness of a given program with respect to a given specification. We outline our INFER algorithm in Fig. 3, which accepts a program and its specification provided as a SyGuS-INV encoding (which is currently restricted to single-loop programs) – a precondition P, a transition function T, a postcondition Q, a set $\mathcal{Z}$ of states, and returns a provably sufficient loop invariant $\mathcal{I}$.

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

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

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, we would start with $\mathcal{I} = ((x < y) \vee (x \geqslant 1))$.

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

However, this candidate invariant is not inductive. The state $(x \mapsto 0 \land y \mapsto 1)$ satisfies $\mathcal{I}$, but it may transition to state $(x \mapsto 0 \land y \mapsto 0)$, which violates $\mathcal{I}$. In lines 2 – 9, INFER employs a *strengthening* loop (inspired by HOLA [4]), to ensure inductiveness of the candidate. In each iteration, it learns a precondition Δ under which the candidate invariant would be preserved after transitions. For our running example, $\Delta = (x \geqslant 1)$, for instance, would ensure inductiveness of our candidate invariant $\mathcal{I} = ((x < y) \lor (x \geqslant 1))$ over all possible transitions. In line 9, we strengthen the candidate invariant by conjoining it with the learned precondition. For our running example, the new candidate $\mathcal{I} = (x \geqslant 1)$ 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, and learns a *likely* precondition Δ 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 – 8, till a provably sufficient precondition is learned.

Once we have an inductive candidate invariant that is stronger than the postcondition, the final property we need to ensure is its weakness relative to the precondition. In line 10, we use CHECK to verify this, and look for a counterexample C. 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, due to non-determinism within the program, for instance. On finding such a program state, we collect a few more (k) states starting with C, in line 12, to account for the unexplored program behavior. Finally, in line 13, we restart with the new set of available program states. Note that if no counterexample C is found, as is the case with the candidate $\mathcal{I} = (x \geqslant 1)$ for our running example, our candidate invariant is provably sufficient.

III. IMPLEMENTATION

Our implementation of LOOPINVGEN is open source, and is available at https://github.com/SaswatPadhi/LoopInvGen. For the various components, LOOPINVGEN 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 SyGuS-COMP 2017.
- 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 formula [8].

[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].

Our SyGuS-COMP 2017 submission is able to achieve a significantly better performance than our original tool [2], due to the following major optimizations:

- 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).

Finally, LOOPINVGEN uses a *conflict group size* [2] of 64 with PIE, overriding the default size of 16.

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 loop invariant inference problem to a series of precondition inference problems, and solves them using PIE which uses a form of program synthesis for synthesizing new features in a targeted manner, as necessary.

ACKNOWLEDGMENT

Thanks to Rahul Sharma for his contributions to our joint prior work on PIE and LOOPINVGEN; Sumit Gulwani and Zachary Kincaid for access to the ESCHER program synthesis tool; Isil Dillig for access to their loop invariant inference benchmarks; and the organizers of SyGuS-COMP for making all the benchmarks and solvers 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.