FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Štěpán Henrych

The PD-KIND algorithm in the Golem SMT solver

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Jan Kofroň, Ph.D.
Study programme: Informatika

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Dedication.

Title: The PD-KIND algorithm in the Golem SMT solver

Author: Štěpán Henrych

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: Use the most precise, shortest sentences that state what problem the thesis addresses, how it is approached, pinpoint the exact result achieved, and describe the applications and significance of the results. Highlight anything novel that was discovered or improved by the thesis. Maximum length is 200 words, but try to fit into 120. Abstracts are often used for deciding if a reviewer will be suitable for the thesis; a well-written abstract thus increases the probability of getting a reviewer who will like the thesis.

Keywords: keyword, key phrase

Název práce: Algoritmus PD-KIND v řešiči Golem

Autor: Štěpán Henrych

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Abstrakt práce přeložte také do češtiny.

Klíčová slova: klíčová slova, klíčové fráze

Contents

1 Introduction

In computer science and software engineering, the idea of software verification is becoming more and more important. Verification checks whether the program or system operates correctly and fulfills the set properties. The goal is to detect bugs during the development process.

One of the frameworks that is becoming popular is the Constrained Horn Clauses (CHC) framework. CHC is a fragment of First Order Logic modulo constraints that captures many program verification problems as constraint solving. The main advantage of CHC is that it separates modeling from solving by translating the program's behavior and properties into constrained language and then using a specialized CHC solver to solve various verification tasks across programming languages by deciding the satisfiability problem of a CHC system.

Golem [1] is one such solver, which integrates the interpolating SMT solver OpenSMT [2]. Golem currently implements six model-checking algorithms to solve the CHC satisfiability problem.

On top of solving the CHC satisfiability problem, each engine in Golem provides a validity witness for their answer. In software verification we can think of these witnesses as invariants for SAFE answers and counterexample path for UNSAFE answer. By providing these witnesses, we can ensure that the engines answer is correct. Also to check, wheter the engine doesn't give false witnesses, Golem has built in an internal validator to verify the correctness of the witness.

Describe PDKIND

1.1 Goals

Our goal in this work is to

2 Definitions

Describe theory needed to understand the following chapters.

2.1 Transition System

2.2 Satisfiability Modulo Theories

2.3 OpenSMT

3 Golem

What Golem is. How it works. What inputs it recieves. How is the framework structured (language etc.). Analyze how we integrate our engine into it (library, different language or the same way as other engines are integrated.

Golem is a satisfiability solver for constrained horn clauses over linear, real, and integer arithmetic. It consists of multiple model-checking algorithms, which solve the satisfiability problem. Currently, there are five of them: BMC, KIND, IMC, LAWI, and SPACER. In this thesis, we will be adding an implementation of the PD-KIND algorithm into the solver.

4 PDKind

In this chapter, in sections 4.1-4.4, we will be using methodology and definitions described in [3].

4.1 Induction vs k-Induction

This algorithm is a combination of IC3 and k-induction. IC3 is a commonly used method that uses induction to show a property is invariant by incrementally constructing an inductive strengthening of the property. PDKind breaks IC3 into modules and that allows replacing the induction method with k-induction.

Definition (Induction): Proves a property P is invariant by showing:

- **Base Case (init)**: P holds in the initial state.
- **Inductive Step (cons)**: If P holds in a state, it holds in the next state.

Definition (k-Induction): Extends traditional induction to consider sequences of k states. Proves a property P is invariant by showing:

- **Base Case (k-init)**: P holds in the first k states.
- **Inductive Step (k-cons)**: If P holds in a sequence of k states, it holds in the next state.

This method is more powerful for properties that are not inductive but can be shown to hold over multiple steps. More precise definitions are shown in [3].

Relative Power

With Quantifier Elimination: Induction and k-induction have the same deductive power. K-Induction might provide more concise proofs.
Without Quantifier Elimination: K-Induction can be exponentially more concise than induction. Stronger in certain logical theories like pure Boolean logic or linear arithmetic.

Practical Effectiveness

[3] shows that k-Induction is effective, especially when combined with algorithms like IC3. Here we show its effectiveness on a simple example.

Consider a transition system with an array a and the following properties:

Invariant Property P: $a[0] = 0$
Initial State:
$i \leftarrow 0$
$j \leftarrow 0$
$a[0] \leftarrow 0$

Transition:

Randomly select j
Increment i
$a[i] \leftarrow a[j]$

Induction cannot prove P as invariant because P is not inductive in a single step. K-Induction, however, can be used to prove P as $(N+1)$-inductive, where N is the length of the array.

Base Case (k-init): P holds in the first k states.
Inductive Step (k-cons): If P holds in a sequence of k states, it holds in the next state.

This shows that k-induction can handle properties where simple induction fails, especially in systems with complex state transitions.

4.2 Rechability checking procedure

To understand the following, we need to introduce a few definitions. Let us have a formula in the form

$$A(\vec{x}) \wedge T[B]^k(\vec{x}, \vec{y}, \vec{w}) \wedge C(\vec{y}) \tag{4.1}$$

Where for $k > 1$, $T[F]^k(\vec{x}, \vec{x}')$ is defined as

$$T(\vec{x}, \vec{w}_1) \wedge \bigwedge_{i=1}^{k-1} (F(\vec{w}_i) \wedge T(\vec{w}_i, \vec{w}_{i+1})) \wedge T(\vec{w}_{k-1}, \vec{x}')$$

where $\vec{w}$ are state variables in the intermediate states.

Definition (Interpolant)[3]: If formula (4.1) is unsatisfiable, then $I(\vec{y})$ is an interpolant if

1. $A(\vec{x}) \wedge T[B]^k(\vec{x}, \vec{w}, \vec{y}) \Rightarrow I(\vec{y})$, and
2. $I(\vec{y})$ and $C(\vec{y})$ are inconsistent.

Definition (Generalization)[3]: If formula (4.1) is satisfiable, then $G(\vec{x})$ is a generalization if

1. $G(\vec{x}) \Rightarrow \exists \vec{y}, \vec{w}\ T[B]^k(\vec{x}, \vec{y}, \vec{w}) \wedge C(\vec{y})$, and
2. $G(\vec{x})$ and $A(\vec{x})$ are consistent.

Algorithm 1 Reachable

```
Input: Target state F, maximum steps k
Data: Reachability frames R, initial states I, transition states T
Output: True if F is reachable in k steps, False otherwise
if k = 0 then
    return CheckSAT(I ∧ T^0 ∧ F)
end if
while true do
    if CheckSAT(R_{k-1} ∧ T ∧ F) then
        G ← Generalize(R_{k-1}, T, F)
        if Reachable(G, k − 1) then
            return true
        else
            E ← Explain(G, k − 1)
            R_{k-1} ← R_{k-1} ∪ E
        end if
    else
        return false
    end if
end while
```

Method shown in Algorithm 1 tries to reach the initial states backwards by using a depth-first search strategy.

To check if F is reachable from the initial states in k steps, we first check whether F is reachable in one transition from the previous frame R_{k-1}. If there is no such transition, then F is not reachable in k steps. Otherwise, we get a state that satisfies R_{k-1} and from which F is reachable in one step. We then call a generalization procedure, which gives us a formula G, a generalization of the state mentioned above. Then, using a DFS strategy, we recursively check whether G is reachable from the initial states. If G is reachable, then F is also reachable, and the procedure ends. Otherwise, we can learn an explanation and eliminate G by adding the explanation into the frame R_{k-1}.

In our implementation, we needed to figure out the following: **Satisfiability checking**, **Generalization**, **Explanation** and **Reachability frames representation**.

Satisfiability Checking

We could use various SMT solvers for satisfiability checking, but Golem already has a wrapper around the OpenSMT solver and uses it in every other engine. Therefore we will use it too for every other satisfiability check, that we will need.

Generalization

Algorithm 2 Generalize

Input: Model M, transition formula T, state formula F
Output: Generalized formula G
$StateVars \leftarrow$ GetStateVars()
$G \leftarrow$ KeepOnly(StateVars, $T \wedge F$, M)
return G

Golem doesn't provide us with the generalization method. Still, it has needed components to create our method, shown in pseudocode Algorithm 2. It eliminates all variables except the state ones, represented as $\vec{x}$ from the formula $T[B]^k(\vec{x}, \vec{w}, \vec{y}) \wedge C(\vec{y})$ to satisfy the generalization definition.

Explanation

Instead of the *Explain()* method, we will use another feature of the OpenSMT solver and that is interpolation. To get the interpolant, we tell the solver to give us an interpolation of the first two formulas inserted in it, i.e. for *CheckSat(*$A \wedge B \wedge C$*)* we want interpolation of $A \wedge B$.

On line 13 in Algorithm 1, we need an interpolant from the *CheckSAT()* that happened in the previous *Reachable()* call. Therefore, we need to modify our *Reachable()* method on line 17 to also return an interpolant along with the false result. Later, we will see that in our implementation each *CheckSAT()* has its solver instance. To get the interpolant, we just ask the solver to return it if the *CheckSAT()* fails.

Reachability frames representation

For the *Reachability frames* representation, we had several choices. The simplest one was to create a list R of formulas (where $R_i := R[i]$) each time we call *Reachable()* from outside. This approach is simple, yet not efficient since we would be losing the whole *Reachability frames* we built in each call, instead of reusing it in another call.

Therefore we will create a better approach and that is a *Reachability checker* class, where each instance of this class will have such list, but calling *Reachable()* on that instance would only grow the *Reachability frames* and wouldn't delete it.

4.3 Push procedure

Definition (Induction Frame)[3]: A set of tuples $F \subset \mathbb{F} \times \mathbb{F}$, where $\mathbb{F}$ is a set of all state formulas in theory T, is an induction frame at index n if $(P, \neg P) \in F$ and $\forall(lemma, counterExample) \in F$:

- $lemma$ is valid up to n steps and refutes $counterExample$
- $counterExample$ states can be extended to a counterexample to P.

This procedure shown in Algorithm 3 is the core of PDKind algorithm, We will break it down into smaller pieces to understand how it works.

The first part starts on line 11 where we need to check if lemma is k-inductive. We can also notice that the CheckSAT returns model m_1. In our implementation, CheckSAT doesn't return model, but we can solve this by initializing a new solver instance for each check. If the check is true, we can ask the solver for a model that satisfies the inserted formula. If the check isn't successful, we can push a new obligation to our new induction frame G and continue. Else we get the model m_1 and save it for later.

In the second part, on line 16, we check if $counterExample$ is reachable. If it is, we get model m_2 and generalize it to g_2. We know that from g_2, we can reach $\neg P$, so we need to check if g_2 is reachable from initial states. If it is reachable, the property is invalid, and we mark $isInvalid \leftarrow true$.

On line 19, we can see that *Reachable()* accepts more arguments and returns more values than shown in Algorithm 1. This new *Reachable(i,j,F)* method checks if F is reachable in k steps where $i \leq k \leq j$. To achieve this behavior in our implementation, we create a wrapper function that calls *Reachable(k, F)* in a for loop and returns the first k where the call was successful along with the result. The i return value is an interpolant of the last *Reachable()* check if the whole check wasn't successful. If this call wasn't successful, we get an interpolant i_1 and assign it to g_3 which eliminates g_2. We found a new induction obligation (g_3, g_2), which is a strengthening of F. Now, we can try again with a potential counterexample eliminated.

Last step is to analyze the induction failure. From the first check we have a model m_1, which is a counterexample to the k-inductiveness of $lemma$. We again get g_1 as a generalization of m_1 and check if g_1 is reachable from initial states. If it is reachable, we replace $lemma$ with weaker $\neg counterExample$ and push this new obligation to F and G. On the other hand, if g_1 is not reachable, we strengthen $lemma$ with g_3 and push this new obligation to F.

Algorithm 3 Push

Input: Induction frame F, n, k
Output: Old induction frame F, new induction frame G, n_p, $isInvalid$
push elements of F to queue Q
$G \leftarrow \{\}$
$n_p \leftarrow n + k$
$invalid \leftarrow false$
while $\neg invalid$ and Q is not empty **do**
  $(lemma, counterExample) \leftarrow Q$.pop()
  $F_{ABS} \leftarrow \bigwedge a_i$,where $(a_i,\, b_i) \in F\ \forall i \in \{1,...,F$.length()$\}$
  $T_k \leftarrow T[F_{ABS}]^k$ by definition
  $(s_1, m_1) \leftarrow$ CheckSAT$(F_{ABS}, T_k, \neg lemma)$ // m_1 is model if is SAT
  if $\neg s_1$ **then**
    $G \leftarrow G \cup (lemma, counterExample)$
    Continue
  end if
  $(s_2, m_2) \leftarrow$ CheckSAT$(F_{ABS} \wedge T_k \wedge counterExample)$
  if s_2 **then**
    $g_2 \leftarrow$ Generalize$(m_2, T_k, counterExample)$
    $(r_1, i_1, n_1) \leftarrow$ Reachable$(n - k + 1,\, n,\, g_2)$ // i_1 is interpolant
    if r_1 **then**
      $isInvalid \leftarrow true$
      Continue
    else
      $g_3 \leftarrow i_1$
      $F \leftarrow F \cup (g_3, g_2)$
      Q.push$((g_3, g_2))$
      Q.push$(lemma, counterExample)$
      Continue
    end if
  end if
  $g_1 \leftarrow$ Generalize$(m_1,\, T_k,\, \neg lemma)$
  $(r_2, i_2, n_2) \leftarrow$ Reachable$(n - k + 1,\, n,\, g_1)$
  if r_2 **then**
    $(r_3, i_3, n_3) \leftarrow$ Reachable$(n + 1,\, n_2 + k,\, g_1)$
    $n_p \leftarrow$ Min$(n_p,\, n_3)$
    $F \leftarrow F \cup (\neg counterExample,\, counterExample)$
    $G \leftarrow G \cup (\neg counterExample,\, counterExample)$
  else
    $g_3 \leftarrow i_2 \wedge lemma$
    $F \leftarrow F \cup (g_3,\, counterExample)$
    $F \leftarrow F \setminus (lemma,\, counterExample)$
    Q.push$((g_3,\, counterExample))$
  end if
  return $(F,\, G,\, n_p,\, isInvalid)$
end while

4.4 PD-Kind procedure

The main PDKind procedure shown in Algorithm 4 checks if P is invariant by iteratively calling the Push procedure to find a k-inductive strengthening of P for some $1 \leq k \leq n+1$. The strengthening G is k-inductive and if $F = G$, then P is invariant and we return SAFE. If the Push procedure marks isInvalid as true, the property is not invariant and we return UNSAFE. Otherwise, we update n and repeat the loop.

In our implementation, the property P is a negation of a query, that we get on the input, which represents bad states. We also need to check if the initial states are empty, which would result in SAFE, or if the query holds in the initial states, which would result in UNSAFE.

Algorithm 4 Main PD-Kind procedure

Input: Initial states I, transition formula T, property P
Output: Retrun UNSAFE if P is invalid or SAFE when there is no inductive strengthening left
$n \leftarrow 0$
$F \leftarrow (P, \neg P)$
while true **do**
    $k \leftarrow n + 1$
    $(F, G, n_p, isInvalid) \leftarrow$ Push(F, n, k)
    if $isInvalid$ **then**
        return UNSAFE
    end if
    if $F = G$ **then**
        return SAFE
    end if
    $n \leftarrow n_p$
    $F \leftarrow G$
end while

4.5 Validity checking

In many cases, it is often required to provide a witness to the answer obtained from solving the CHC satisfiability problem. In software verification, a satisfiability witness corresponds to a program invariant, and an unsatisfiability witness corresponds to counterexample paths. Generally, a satisfiability witness is a model that provides an interpretation of all CHC predicates and variables that satisfy all the clauses. An unsatisfiability witness is a proof presented as a sequence of derivations of ground instances of the predicates, where for the proof to be valid, each premise must be a conclusion of some previously derived step.

In Golem [1], each engine provides a validity witness when the option `--print-witness` is used. To follow the structure Golem has, we need to implement such an option for our PDKind engine as well.

UNSAT witness

First, we will describe the implementation of the unsatisfiability witness in our engine. The goal is to generate paths to counterexamples during the CHC satisfiability solving process. To do that, we utilize a function in the Golem solver, which can generate the path to a counterexample based on the number of steps required to reach the counterexample, for clarity, we will call them steps to the counterexample. This allows us to only keep track of the steps to counterexample for each counterexample we encounter during the CHC solving. To do this, we take the *Induction frame* (*lemma*, *counterExample*) and create a structure for the *counterExample*, which will hold the formula and the steps to the counterxample.

Next, we need to correctly assign the steps to the counterexample to each counterexample, that we find. In Algorithm 3, we note that a new counterexample g_2 is created only on line 18, which we then use in the else branch starting on line 23. The steps to counterexample assigned to g_2 is the steps to the counterexample of $counterExample + k$, which is the length of the transition passed as a parameter to *Generalize()* on line 16.

Then we need to modify *Push()* to return the steps to the counterexample. On line 21, when we encounter a reachable counterexample g_2, we assign the steps to the counterexample returned by *Push()* to be the steps to the counterexample of g_2.

SAT witness

In this section, we will describe the implementation of the satisfiability witness in the PDKind engine. The aim is to construct a program invariant during the CHC satisfiability solving process.

During the *Push()* procedure described in Algorithm 3, we are building an *Induction Frame*, which is a set of tuples (*lemma*, *counterExample*), where the lemma holds for n steps and refutes the counterExample. After the solving procedure is finished, we end up on line 12 of the PDKind procedure in Algorithm 4, because we are constructing the satisfiability witness. We can now take the final *Induction Frame* and make a conjunction of all lemmas. What we get is an n-inductive invariant because the lemmas hold for n steps and there exists no strengthening of the *Induction Frame*, as on line 11 we get that $F = G$.

The final step is to transform the n-inductive invariant into an inductive invariant. To do that, we utilize the Golem solver's function *kinductiveToInductive()*, which takes the n-inductive invariant, n, and the system and returns an inductive invariant. This invariant is the validity witness for the SAT answer.

During the development we could validate those witnesses by using Golem solver's inner validator, which takes the witness and validates it. For that we created various tests, testing both the correctness of the engine and the correctness of the witnesses.

5 Implementation

Describe the API, the main functions of the engine, its structure and how we used other parts of Golem.

5.1 Reachability class

5.1.1 reachable

5.1.2 checkReachability

5.2 PD-Kind engine

5.2.1 solve

5.2.2 push

5.2.3 generalize

5.3 Data structures

5.3.1 Reachability frame

5.3.2 Induction frame

6 Experiments

Compare PDKind with other engines in Golem. Possibly with other solvers.

I also noticed that there is a part in the code where we are supposed to pick a number between k_1 and k_2. So far it always picks k_1. We could make more approaches and compare them.

7 Conclusion

Bibliography

1. Blicha, Martin; Britikov, Konstantin; Sharygina, Natasha. The Golem Horn Solver. In: Enea, Constantin; Lal, Akash (eds.). *Computer Aided Verification*. Cham: Springer Nature Switzerland, 2023, pp. 209–223. Lecture Notes in Computer Science. isbn 978-3-031-37703-7. Available from doi: 10.1007/978-3-031-37703-7_10.
2. Hyvärinen, Antti E. J.; Marescotti, Matteo; Alt, Leonardo; Sharygina, Natasha. OpenSMT2: An SMT Solver for Multi-core and Cloud Computing. In: Creignou, Nadia; Le Berre, Daniel (eds.). *Theory and Applications of Satisfiability Testing – SAT 2016*. Cham: Springer International Publishing, 2016, pp. 547–553. isbn 978-3-319-40970-2.
3. Jovanović, Dejan; Dutertre, Bruno. Property-directed k-induction. In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 2016, pp. 85–92. Available from doi: 10.1109/FMCAD.2016.7886665.

List of Figures

List of Tables

List of Abbreviations

A Attachments

A.1 First Attachment