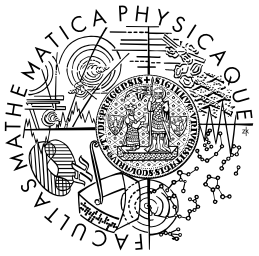

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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

1.1 Goals

Describe the goals of this work, what we added and what we expect.

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

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 [1].

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

[1] 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): 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): 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

```
1: Input: Target state F, maximum steps k
2: Data: Reachability frame R, initial states I, transition states T
3: Output: True if F is reachable in k steps, False otherwise
4: if k = 0 then
5:     return CheckSAT(I ∧ T^0 ∧ F)
6: end if
7: while true do
8:     if CheckSAT(R_{k−1} ∧ T ∧ F) then
9:         G ← Generalize(R_{k−1}, T, F)
10:        if Reachable(G, k − 1) then
11:            return true
12:        else
13:            E ← Explain(G, k − 1)
14:            R_{k−1} ← R_{k−1} ∪ E
15:        end if
16:    else
17:        return false
18:    end if
19: 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 **R frame 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

1: **Input:** Model M, transition formula T, state formula F
2: **Output:** Generalized formula G
3: $StateVars \leftarrow$ GetStateVars()
4: $G \leftarrow$ KeepOnly(StateVars, $T \wedge F$, M)
5: **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 SATCheck has its solver instance. To get the interpolant, we just ask the solver to return it if the SATCheck fails.

R Frame representation

For the R Frame 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 R Frame we built in each call, instead of reusing it in another call.

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

4.3 Push procedure

Definition (Induction Frame): 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.

Algorithm 3 Push

1: **Input:** Induction frame F, n, k
2: **Output:** Old induction frame F, new induction frame G, n_p, $isInvalid$
3: push elements of F to queue Q
4: $G \leftarrow \{\}$
5: $n_p \leftarrow n + k$
6: $invalid \leftarrow false$
7: **while** $\neg invalid$ and Q is not empty **do**
8: $(lemma, counterExample) \leftarrow Q$.pop()
9: $F_{ABS} \leftarrow \bigwedge a_i$,where $(a_i, b_i) \in F\ \forall i \in \{1,...,F.\text{length}()\}$
10: $T_k \leftarrow T[F_{ABS}]^k$ by definition
11: $(s_1, m_1) \leftarrow \text{CheckSAT}(F_{ABS}, T_k, \neg lemma)$ // m_1 is model if is SAT
12: **if** $\neg s_1$ **then**
13: $G \leftarrow G \cup (lemma, counterExample)$
14: Continue
15: **end if**
16: $(s_2, m_2) \leftarrow \text{CheckSAT}(F_{ABS} \wedge T_k \wedge counterExample)$
17: **if** s_2 **then**
18: $g_2 \leftarrow \text{Generalize}(m_2, T_k, counterExample)$
19: $(r_1, i_1, n_1) \leftarrow \text{Reachable}(n - k + 1,\ n,\ g_2)$ // i_1 is interpolant
20: **if** r_1 **then**
21: $isInvalid \leftarrow true$
22: Continue
23: **else**
24: $g_3 \leftarrow i_1$
25: $F \leftarrow F \cup (g_3, g_2)$
26: Q.push$((g_3, g_2))$
27: Q.push$(lemma, counterExample)$
28: Continue
29: **end if**
30: **end if**
31: $g_1 \leftarrow \text{Generalize}(m_1,\ T_k,\ \neg lemma)$
32: $(r_2, i_2, n_2) \leftarrow \text{Reachable}(n - k + 1,\ n,\ g_1)$
33: **if** r_2 **then**
34: $(r_3, i_3, n_3) \leftarrow \text{Reachable}(n + 1,\ n_2 + k,\ g_1)$
35: $n_p \leftarrow \text{Min}(n_p,\ n_3)$
36: $F \leftarrow F \cup (\neg counterExample,\ counterExample)$
37: $G \leftarrow G \cup (\neg counterExample,\ counterExample)$
38: **else**
39: $g_3 \leftarrow i_2 \wedge lemma$
40: $F \leftarrow F \cup (g_3,\ counterExample)$
41: $F \leftarrow F \setminus (lemma,\ counterExample)$
42: Q.push$((g_3,\ counterExample))$
43: **end if**
44: **return** $(F,\ G,\ n_p,\ isInvalid)$
45: **end while**

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 SAT check returns model m_1. In our implementation, SAT checking doesn't return model, but we can solve this by initializing a new solver instance for each SAT 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.

4.4 PD-Kind procedure

The main PDKind procedure shown in Algorith 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

1: **Input:** Initial states I, transition formula T, property P
2: **Output:** Retrun UNSAFE if P is invalid or SAFE when there is no inductive strenghtening left
3: $n \leftarrow 0$
4: $F \leftarrow (P, \neg P)$
5: **while** true **do**
6: $k \leftarrow n + 1$
7: $(F, G, n_p, isInvalid) \leftarrow \text{Push}(F, n, k)$
8: **if** $isInvalid$ **then**
9: **return** UNSAFE
10: **end if**
11: **if** $F = G$ **then**
12: **return** SAFE
13: **end if**
14: $n \leftarrow n_p$
15: $F \leftarrow G$
16: **end while**

4.5 Validity checking

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. 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