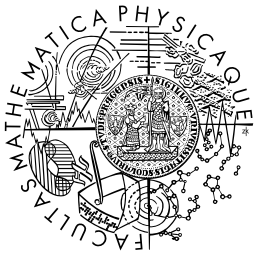

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 given properties. The goal is to detect bugs and errors during the development process.

One of the frameworks that is becoming popular is the Constrained Horn Clauses(CHC)[6] 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[3] is one such solver, which integrates the interpolating SMT solver OpenSMT[7]. 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 a path to a counterexample 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.

One such algorithm, that can be used to solve these problems is PDKind (Property-Directed K-induction)[8]. PDKind is an IC3[13]-based algorithm, that separates reachability checking and induction reasoning, allowing to replace the induction core with k-induction.

1.1 Goals

The goal of this work is to create a PDKind engine and integrate it into the Golem Horn Solver[3].

We will start with defining the background around SMT solving. Then, we will introduce the Golem solver and analyze the PDKind algorithm to design a solution that integrates the PDKind algorithm as an engine into the Golem solver. Afterward, we will implement this design and add the generation of validity witnesses to the implementation. We will test the correctness and efficiency of the implementation using a set of benchmarks and compare the results with other engines. The correctness of the witnesses will be verified using a set of tests that utilize an internal function of the Golem solver, which can verify the correctness of the witnesses.

2 Background

2.1 Transition System

In computer science and software verification, it is often helpful to use transition systems as a layer of abstraction over various problems, such as the satisfiability of the Constrained Horn Clause (CHC). Transition systems make it easier to analyze and verify given properties by providing a way to represent the behavior of a computer system or some abstract computation device. This chapter introduces the basic concepts of Transitional Systems, which will help us understand the following chapters.

Definition (Transition System)[8]: A transition system is a 4-tuple (S, $\mathcal{F}$, $\mathcal{T}$, $\mathcal{I}$), where:

- S is a finite set of state variables $\vec{x}$, where each $x \in \vec{x}$ has its primed version x' and state s is an interpretation of $\vec{x}$ that assigns a value $s(x)$ to each $x \in \vec{x}$.
- $\mathcal{F}$ is a set of quantifier-free formulas, where for any $F \in \mathcal{F}$ and a state variable $\vec{x}$, $F(\vec{x})$ is a state formula and holds in a state s.
- $\mathcal{T}$ is a set of quantifier-free formulas, where for any $T \in \mathcal{T}$ and a state variable $\vec{x}$, $T(\vec{x}, \vec{x}')$ is a transition formula describing transitions.
- $\mathcal{I}$ is a set of formulas, where for any $I \in \mathcal{I}$ and a state variable $\vec{x}$, $I(\vec{x})$ is a state formula describing initial states.

Definition (Successor):[8] State s has a successor s', when $T(s(\vec{x}), s'(\vec{x}'))$ is true.

Definition (k-reachability):[8] State s is k-reachable if there exists a sequence $I = s_0, s_1, s_2, \ldots, s_k = s$, where each s_{i+1} is a successor of s_i.

Let us have a transition system and property P, then P is invariant if all states reachable in the system satisfy it. Otherwise, there exists a trace to a counterexample $\neg P$.

2.2 Satisfiability Modulo Theories

In logic and computer science, the Boolean satisfiability problem, also called SAT, is one of the most well-known problems. In 1971, it was proven to be the first NP-complete problem [5]. This greatly helped in the field of complexity theory by providing a tool for the classification of other computational problems. SAT is widely used, but there are cases where it is not possible to represent a problem using only the two Boolean values, true and false.

This led to the generalization of SAT to more complex formulas, which can include real numbers, integers, or even lists and other data structures. In other

words, an SMT instance arises as a generalization of SAT instance, where Boolean variables are replaced with predicates of some theory. Examples of such theories are uninterpreted functions (UF), arrays or linear arithmetics (LA), or, more specifically, linear real arithmetics (LRA), which we will focus on in the rest of this work.

2.3 OpenSMT

OpenSMT[4] is an open-source SMT solver developed by the Formal Verification and Security Lab based at the University of Lugano. It includes a parser for SMT-LIB[1] language, a state-of-the-art SAT-Solver, and a clean interface for new theory solvers, currently supporting various logics such as QF_UF, QF_LIA, QF_LRA, and more. Currently, there is a second version OpenSMT2[7], which supports interpolation for propositional logic, QF_UF, QF_LRA, QF_LIA, and QF_IDL.

2.4 SMT-LIB

SMT-LIB[1] is an international initiative that provides standard descriptions of background theories used in SMT solvers and develops common input and output languages for SMT solvers. It offers a unified framework for the description of SMT problems. Additionally, SMT-LIB maintains a library of input problems, or benchmarks, written in the SMT-LIB language to support the development of different SMT solvers.

```
; Integer arithmetic
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)

; Assertion
(assert (= (- x y) (+ x (- y) 1)))

; Check satisfiability
(check-sat)

(exit)
```

Figure 2.1 SMT-LIB input example

In the Figure 4.1 we can see an example of the SMT-LIB input taken from the offical examples of SMT-LIB[1] .

- Lines starting with ';' represent comments and are ommited.
- **(set-logic QF_LIA)**: This command sets the logic to Quantifier-Free Linear Integer Arithmetic (QF_LIA), which involves linear integer arithmetic expressions without quantifiers.

- **(declare-const x Int) and (declare-const y Int)**: These commands declare two integer constants x and y.
- **(assert (= (- x y) (+ x (- y) 1)))**: This assertion states that $x - y$ should be equal to $x - y + 1$
- **(check-sat)**: This command asks the SMT solver to check the satisfiability of the assertion.
- **(exit)**: This command exits the solver.

3 Golem

Golem [3] is a flexible and efficient solver for CHC satisfiability problems over linear real arithmetic (LRA) and linear integer arithmetic (LIA), written in C++.

Golem integrates an interpolating SMT solver OpenSMT[7], and currently implements six different back-end engines for CHC solving, where each engine can use the OpenSMT not only for SMT queries but also for interpolant computation.

3.1 Solver overview

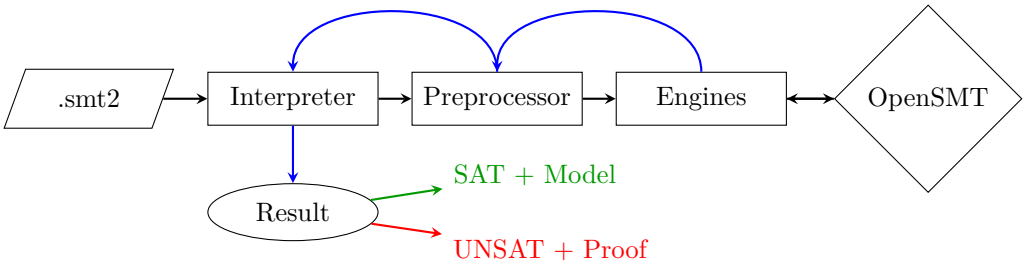

Figure 3.1 Architecture of Golem

In this section, we will describe the Golem solving process depicted in Figure 3.1.

Reading and interpreting CHCs

Golem reads the input from a file in the .smt2 format, which is an extension of the SMT-LIB language[1]. The interpreter builds an internal representation of the CHC system by first normalizing the CHCs to ensure that each predicate has only variables as arguments and then converting the CHCs to the graph representation. The graph representation is then passed to the preprocessor.

Preprocessing

The Preprocessor applies transformations to simplify the graph representation:

- **Predicate Elimination:** Removes predicates, that are not present in both the body and the head of the same clause.
- **Clause Merging:** Merges clauses with the same uninterpreted predicate by disjoining their constraints.
- **Redundant Clause Deletion:** Removes clauses, that cannot participate in the proof of unsatisfiability.

Engines:

The graph is then solved with one of the engines (this option is specified by the user):

- Bounded Model Checking (BMC) [2]
- k-Induction (KIND) [12]
- Interpolation-based Model Checking (IMC) [10]
- Lazy Abstractions with Interpolants (LAWI) [11]
- Spacer [9]
- Transition Power Abstraction (TPA) [3]

The user can also select the option to produce a validity witness. When the engine solves the problem, it generates a model for the SAT result or a proof for the UNSAT result. These models and proofs are translated back by the preprocessor to match the original system.

$$\begin{array}{rcll} x \leq 1 & \implies & I(x) & I(1) \\ x' = x + 1 & \implies & T(x, x') & T(1, 2) \\ I(x) \wedge T(x, x') & \implies & S(x') & S(2) \\ S(x) \wedge x \geq 2 & \implies & \text{false} & \text{false} \end{array}$$

Figure 3.2 UNSAT + Proof example

In Figure 3.2, we have a CHC system and a proof of its unsatisfiability. There are four derivation steps. The first step sets the variable x to 1 and gets I(1). The second step sets $x := 1$ and $x' := 2$ and gets T(1, 2). Step three applies resolution to the instance of the third clause for $x := 1$ and $x' := 2$ and the previously derived facts L(1) and T(1,2) and gets S(2). The last step again applies resolution to the instance of the fourth clause with $x := 2$ and the derived fact S(2) and gets false.

3.2 Engine integration

The architecture described above allows us to integrate an engine into the Golem solver without modifying it. There are several ways we can do it.

Writing a library in a different programming language could be interesting because we could leverage the pros of other languages. The downside is that the interaction of C++ and another language could bring various overheads. Our goal is to create an efficient engine, and therefore, having these overheads would not be optimal.

Another option would be to write a C++ library and include it in the project. This would eliminate the downsides of the first option, but there would still

be some issues. For example, we would have to integrate an SMT solver for satisfiability checking and interpolation.

The most efficient option is to integrate the engine directly into the solver the same way the other engines are integrated. This would allow us to directly utilize OpenSMT, which is integrated into Golem, for satisfiability checking and interpolation. We could also use many other features that Golem has, making our development easier.

4 PDKind

In this chapter, we describe and analyze the PDKind algorithm. In sections 4.1-4.4, we will be using methodology, algorithms, and definitions described in [8]. We will also describe the modifications of the algorithm and the decisions we made when implementing some abstract data structures from the paper.

After reading this chapter, the reader should be able to understand the parts of the PDKind algorithm and orientate in our implementation.

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 [8].

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

[8] shows that k-Induction is effective, especially when combined with algorithms like IC3. Here we demonstrate its effectiveness on a simple example from the paper[8].

```
Invariant Property P: a[0] == 0

Initial States:
i = 0
j = 0
a[0] = 0

Transition:
j = Random() mod (i+1)
i = i + 1 mod N
a[i] = a[j]
```

Figure 4.1 Induction vs K-induction example

Consider a transition system depicted in Figure 4.1. We can see, that induction cannot prove P as invariant because P is not inductive in a single step. P is even k-inductive for $k \leq N$. However, we can use k-induction to prove P is $(N+1)$-inductive, where N is the length of the array.

Proof:

- Any sequence of $N+1$ states has a transition that resets i to 0.
- All transitions increment i to at most $N-1$.
- Pick j between 0 and i and do a[i] = a[j].
- If P holds in all states in this sequence and $i = N-1$ in the last state, then $a[0] = a[1] = \cdots = a[N-1] = 0$.
- If $i \neq N-1$ in the last state, then a[0] is unchanged in next transition.

This concludes that P is $(N+1)$-inductive.

□

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

$$T(\vec{x}, \vec{w}_1) \wedge \bigwedge_{i=1}^{k-2} (F(\vec{w}_i) \wedge T(\vec{w}_i, \vec{w}_{i+1})) \wedge F(\vec{w}_{k-1}) \wedge T(\vec{w}_{k-1}, \vec{x}')$$

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

Definition (Interpolant)[8]: 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)[8]: 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.

```
1: Input: Target state F, maximum steps k
2: Data: Reachability frames 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
```

Algorithm 4.2 Reachable method

Method shown in Algorithm 4.2 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

Golem doesn't provide us with the generalization method. Still, it has needed components to create our method, as shown in pseudocode Algorithm 4.3. 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.

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

Algorithm 4.3 Generalize method

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 4.2, 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` class, where each instance of this class will hold 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)[8]: 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.

The procedure shown in Algorithm 4.4 is the core of the 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 do that by checking if the formula $(F_{ABS} \wedge T_k \wedge \neg lemma)$ is satisfiable. We can notice that the `CheckSAT` call returns model `m_1`. In our implementation, `CheckSAT` doesn't directly return model, but we can ask the solver to provide one. This is done by initializing a new solver instance for each check and then asking for a model, if the check is true. If the formula $(F_{ABS} \wedge T_k \wedge \neg lemma)$ isn't satisfiable, 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 ¬`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` ←`true`.

On line 19, we can see that `Reachable()` accepts more arguments and returns more values than shown in Algorithm 4.2. 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 ¬`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`.

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: $(s_1, m_1) \leftarrow \text{CheckSAT}(F_{ABS}, T_k, \neg lemma)$
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: $(r_1, i_1, n_1) \leftarrow \text{Reachable}(n - k + 1,\ n,\ g_2)$
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**

Algorithm 4.4 Push procedure

4.4 PD-Kind procedure

The main PDKind procedure shown in Algorithm 4.5 checks if propert `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`.

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 strengthening left
3: $n \leftarrow 0$
4: $F \leftarrow (P, \neg P)$
5: **while** true **do**
6: $k \leftarrow n + 1$
7: $(F, G, n_p, isInvalid) \leftarrow$ 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**

Algorithm 4.5 Main PD-Kind procedure

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 [3], 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 a counterexample` for each potential counterexample we encounter during the CHC solving. The `steps to a counterexample` of a potential counterexample is a number of steps needed to reach the counterexample from the potential counterexample. 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 a counterxample`.

In the next step, we need to correctly assign the `steps to a counterexample` to each counterexample, that we find. In Algorithm 4.4, 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 a counterexample` assigned to `g_2` is the `steps to a counterexample` of `counterExample` increased by `k`.

Finally, we need to modify `Push()` to return the `steps to a counterexample`. On line 21, when we encounter a reachable counterexample `g_2`, we assign the `steps to a counterexample` returned by `Push()` to be the `steps to a 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 an inductive invariant during the CHC satisfiability solving process.

In the `Push()` procedure described in Algorithm 4.4, we are constructing 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 4.5 because we are constructing the satisfiability witness. We can then take the final `Induction Frame` and form a conjunction of all the lemmas within it. This gives us an n-inductive invariant, as the lemmas hold for `n` steps, and there is no other strengthening of the `Induction Frame`, as indicated by $F = G$ at line 11.

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.

5 Implementation

In this chapter, we will be covering the implementation of our solution in more detail. We will analyze the structure, the individual parts of the program and decisions that had to be made. We will also describe the integration of the engine into the Golem solver.

To keep the Golem's structure of engines, we put our solution in files `PDKind.cpp` and `PDKind.h` and didn't create separate files for each part of the solution.

5.1 Data structures

In this section, we will describe the most used data structures in our solution.

5.1.1 Golem structures

PTRef

The main data structure used in Golem is `PTRef` from OpenSMT. `PTRef` is a reference structure that points to another structure, representing a single term in a formula called `PTerm`. `PTRef` is just a number as an identifier to differentiate between the terms. The mapping between `PTRef` and `PTerm` is handled by the `Logic` class, respectively, one of its implementations. The `Logic` class keeps a mapping table between the `PTRef` and `PTerm`. It also provides methods that we can use to create new terms. Each implementation of the `Logic` class also has functions according to the theory it uses. In our solution, we use the QF_LRA theory. Therefore, we will be using this theory in the OpenSMT too.

We don't need to delve into much more detail here because, in our solution, we will be directly using only `PTRef` and some basic functions of the `Logic` class, shown in Figure 5.1.

```
1 PTRef x = logic.mkIntVar("x");
2 PTRef y = logic.mkIntVar("y");
3
4 PTRef formula = logic.mkAnd(
5     logic.mkEq(x, logic.getTerm_IntOne),
6     logic.mkEq(y, (logic.mkPlus(x, logic.getTerm_IntOne))));
```

Figure 5.1 PTRef and Logic usage example

The first two lines initiate `x` and `y` as an integer variable. On the fourth line, we use the `Logic` class to create formulas $(x = 1)$ and $(y = x + 1)$, and then we use it again to get a conjunction of these formulas and store it in the `PTRef formula`.

MainSolver

Another structure that we will use is the `MainSolver` class from OpenSMT. `MainSolver` will serve us as the main solver for satisfiability checking, model

generation, and interpolation. The solver is initialized with a config.

In the config, we can, for example, specify, if we need to generate interpolants because the solver only produces models by default. In Figure 5.2, we can see an example of the `MainSolver` usage with model generation and interpolation.

```
1  SMTConfig config;
2  config.setOption(SMTConfig::o_produce_inter, SMTOption(true), "ok");
3  config.setSimplifyInterpolant(4);
4
5  MainSolver solver (logic, config, "Example solver");
6  solver.insertFormula(A);
7  solver.insertFormula(B);
8  solver.insertFormula(C);
9
10 sstat result = solver.check();
11 if(result == s_True) {
12     std::unique_ptr<Model> model = solver.getModel();
13     // Do something with the model...
14 } else {
15     auto itpContext = solver.getInterpolationContext();
16     vec<PTRef> itps;
17     int mask = 3;
18     itpContext->getSingleInterpolant(itps, mask);
19     assert(itps.size() == 1);
20     PTRef interpolant = itps[0];
21     // Do something with the interpolant...
22 }
```

Figure 5.2 MainSolver usage example

On lines 1-3, we initialize the config for the solver. The function on line number 3 chooses an interpolating algorithm with the value 4. On lines 5-8, we initiate the solver and insert three formulas, A, B, and C. Lines 10-13 are pretty simple. We call the `check()` procedure, and if the result is `s_True`, we get the model.

Othwerwise, we can get the interpolant. The crucial part is choosing the correct mask. The mask represents which formulas in the solver should be included in the interpolation. In the binary representation of the mask, the i-th bit corresponds to an i-th formula in the solver frame. A bit of value 1 includes the corresponding formula, and 0 doesn't. In our case, the mask is equal to 3 (in binary 011). Therefore, we will create an interpolant of $A \wedge B$ because we include A and B and exclude C.

5.1.2 PDKind structures

Reachability frames

As mentioned in the paper[8], we need to create a structure that can hold formulas that represent k-reachable states from the initial states for all $k \in \{1, \ldots, n\}$. We also need to be able to access the k-reachable formula with the argument `k` and update the formula. To do that, we create a structure `RFrames`, which keeps a vector `r` of formulas `PTRef`, where $r[i] := i$-reachable states from initial states.

Inserting a formula `f` is creating a conjunction of `f` and `r[i]` and storing it in `r[i]`. As shown in Figure 5.3, we overloaded the `[]` operator to allow quick

access to the vector. Also, if `r[i]` doesn't exist, we fill the vector from `r.size()` up to `i` with the term `true`. Before updating the i-th frame, the insert method fills up the vector too, if `r[i]` doesn't exist.

```
class RFrames {
    std::vector<PTRef> r;
    Logic & logic;
public:
    RFrames(Logic & logic) : logic(logic) {}

    PTRef operator[] (size_t i) {
        if (i >= r.size()) {
            while (r.size() <= i) {
                r.push_back(logic.getTerm_true());
            }
        }
        return r[i];
    }

    void insert(PTRef fla, size_t k) {
        if (k >= r.size()) {
            while (r.size() <= k) {
                r.push_back(logic.getTerm_true());
            }
        }
        PTRef new_fla = logic.mkAnd(r[k], fla);
        r[k] = new_fla;
    }
};
```

Figure 5.3 RFrames structure

Induction frame

The induction frame is another structure mentioned in the paper[8] and defined in the section 4.3. In our solution, we will use `InductionFrame` as a set of `IFrameElement`.

The `IFrameElement` is a structure holding a `PTRef lemma` and a `CounterExample counter_example`. In our solution, we keep the `counter_example` in a `CounterExample` structure instead of the `PTRef` because we need to store other data along with the formula. In the `CounterExample` structure, we hold the `PTRef counter_example` formula and also a number of steps needed to reach the `counter_example` from the initial states. We keep this information for the production of the unsatisfiability witness.

5.2 PDKind architecture

In this section, we will describe the functions of the PDKind engine and how they interact together with the OpenSMT solver, as shown in Figure 5.4.

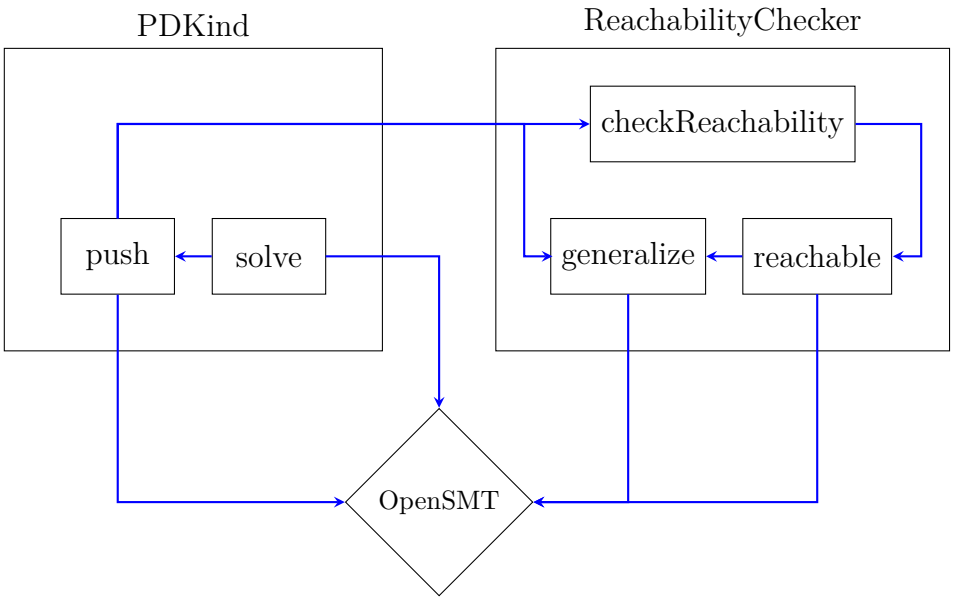

Figure 5.4 Architecture of PDKind engine

5.2.1 ReachabilityChecker class

```
1  class ReachabilityChecker {
2  private:
3      RFrames r_frames;
4      Logic & logic;
5      TransitionSystem const & system;
6      std::tuple<bool, PTRef> reachable(unsigned k, PTRef formula);
7  public:
8      ReachabilityChecker(Logic & logic, TransitionSystem const &
       system) : r_frames(logic), logic(logic), system(system) {}
9      std::tuple<bool, int, PTRef> checkReachability(int from, int to,
        PTRef formula);
10     PTRef generalize(Model & model, PTRef transition, PTRef formula)
       ;
11 };
```

Figure 5.5 ReachabilityChecker class

The `ReachabilityChecker` class encapsulates the mechanism needed for reachability checking. As we analyzed in Section 4.2, using the structure `RFrames` wouldn't be efficient. Therefore, we keep one instance of the `RFrames` in the `ReachabilityChecker` and use it for all the reachability checks.

The `ReachabilityChecker` class could have its file, but we respect the structure of Golem, and we put it in the engine file together with other code.

reachable

The `reachable(unsigned k, PTRef formula)` function is the core of the `ReachabilityChecker` class. The function implements the pseudocode shown in

Algorithm 4.2. But for the function to work correctly with the rest of the engine, we need to modify it.

At first, we also need to return a formula with the reachability result. If the `reachable()` function ends up `False`, we ask the solver to produce an interpolant and return it with the `False` result. Otherwise, we return `True` and a false term.

To produce the interpolants, we must set up a config at the beginning of the function. Later, we will use that config to initialize solvers for satisfiability checking and interpolant production.

The first such solver is used to check whether the given formula holds in the initial states if the given `k` is equal to 0. If the check gives us a false result, we continue to produce the interpolant. We do that the way as shown in Figure 5.2 but, we will use `mask = 1` because we don't insert the transition formula into the solver and therefore we only have $A \wedge B$ and want to interpolate A.

If the given `k` is greater than zero, we can begin with the while loop. Before that, we need to change the version of the given `formula` from 0 to 1. To achieve that, we will use a `TimeMachine` and its function `sendFlaThroughTime(PTRef fla, int steps)`, which takes the `formula` and increases its version by the `steps` number.

It's important to maintain this versioning because we are verifying if the `formula` can be reached with a single transition from a certain RFrame. For this reason, the formulas added to the solver must have the structure of $(R[k-1]_0 \wedge T_{0,1} \wedge formula_1)$. We'll later observe that every `R[i]` is consistently versioned to 0.

In Section 4.2, we have already analyzed that the `Explain()` method on line 13 in Algorithm 4.2 is omitted, and instead, we obtain the interpolant from the `Reachable()` function, called on line 10.

The pseudocode currently concludes after completing these steps, but an additional task remains: generating the interpolant. So far, we have only generated the interpolant for cases where `k = 0`. Now, let's describe how to generate the interpolant for other cases. In the while loop, if the solver check fails, we can ask the solver to produce an interpolant in the same form as shown in Figure 5.2. However, it's important to use the `TimeMachine` to send the interpolant back by one step, ensuring it has version 0. This is consistent with our earlier observation that each `R[i]` has version 0, and the interpolant will be utilized to create or update the `R[i]`. But this doesn't create the complete interpolant yet. We also need to verify whether the given `formula` holds in the initial states. If it does, we can return the interpolant. If not, we need to create an interpolant for the initial states in the same way as we did at the beginning of this function when `k = 0`. Afterward, we combine the two interpolants and return their disjunction.

checkReachability

As we already analyzed in Section 4.3, the Push procedure requires the `reachable()` function to check the reachability in a range of steps instead of a certain number of steps. To achieve this, we establish an additional function that takes a range of steps `(k_from, k_to)` and iterates through the `reachable()` function using a for loop from `k_from` to `k_to`. This function returns the first positive outcome `(True)` or the result of the last unsuccessful call, along with the number of steps used in the most recent `reachable()` function call.

generalize

This function is an implementation of the pseudocode described in Algorithm 4.3. It utilizes the `ModelBasedProjection` class in Golem to eliminate non-state variables from a formula.

Even though the `generalize()` method is utilized not only by the `reachable()` function but also by the `push()` procedure, we chose to keep it in the `ReachabilityChecker` class. This decision was made because this class is already responsible for generating interpolants, i.e., explanations, so it made sense to also handle the production of generalizations.

5.2.2 PD-Kind engine

The `PDKind` class inherits from the `Engine` class in Golem. The `Engine` class contains one virtual method `solve(ChcDirectedHyperGraph const & graph)`. In our engine, we need to override this method to solve the transition system using the PDKind algorithm. This method accepts a hypergraph that needs to be solved as a parameter. First, we need to convert the hypergraph to a normal graph. To solve the normal graph, we can overload the `solve(ChcDriectedGraph const & system)` function to work with the normal graph. Within the overloaded function, we can check if the graph is trivial and utilize Golem's `Common` class to solve it using the `solveTrivial()` function. If the graph is non-trivial, we then transform it into a transition system and call PDKind's `solveTransitionSystem(TransitionSystem const & system)` to apply the PDKind algorithm and solve the system.

solveTransitionSystem

This function accepts the parameter `TransitionSystem const & system`, which provides access to the initial states, transition states, and query. The query represents the bad states that we want to avoid in the provided transition system. The goal is either to construct an inductive invariant that guarantees the satisfiability of the transition system or to construct a path to the counterexample that proves the unsatisfiability of the system. The property to which we want to generate the inductive invariant is the negation of the query.

This function implements the pseudocode shown in Algorithm 4.5. Before the main solving process starts, we need to use the `MainSolver` to check if the initial states are empty, which would result in the `SAFE` answer, and to check if the property holds in the initial states, which would result in the `UNSAFE` answer.

After this part, the implementation pretty much follows the pseudocode. The only difference here is that in our implementation, we need to process the additional data we get from the `push()` procedure, and that is the validity witness information. For satisfiability witness, we take the `InductionFrame` from the last `push()` procedure and first turn it into a k-inductive invariant and then utilize Golem's function `kinductiveToInductive()` to turn it into an inductive invariant. For the unsatisfiability witness, we get the `steps_to_ctx` number from the `push()` procedure and store it as an unsatisfiability witness into the `TransitionSystemVerificationResult` and return it.

push

The `push()` procedure which implements the Algorithm 4.4 is the core of our solution. It connects all parts of our engine together to produce some inductive strengthening or find a counterexample.

In the code snippet in Figure 5.6, we show how we created the transition $T[F_{ABS}]^k$ which satisfies the definition shown in section 4.2 .

We utilize the `TimeMachine` and `Logic` structures to generate a conjunction of transitions $t_{0,1} \wedge t_{1,2} \wedge \cdots \wedge t_{k-1,k}$, with versions ranging from 0 to k-1. Additionally, we create a conjunction of formulas $f\_abs_0 \wedge f\_abs_1 \wedge \cdots \wedge f\_abs_{k-1}$, also versioned from 0 to k-1. Finally, we make a conjunction of these two conjunctions to obtain the final transition formula.

```
1  PTRef t_k = transition;
2  PTRef f_abs_conj = logic.getTerm_true();
3  std::size_t i;
4  for (i = 1; i < k; ++i) {
5      PTRef versionedFla = tm.sendFlaThroughTime(iframe_abs, i);
6      PTRef versionedTransition = tm.sendFlaThroughTime(transition, i)
7      t_k = logic.mkAnd(t_k, versionedTransition);
8      f_abs_conj = logic.mkAnd(f_abs_conj, versionedFla);
9  }
10
11 PTRef t_k_constr = logic.mkAnd(t_k, f_abs_conj);
```

Figure 5.6 Transition creation

The rest of the implementation follows the pseudocode but, we added some parts to enable the witness production.

First, we needed to extend the return values by an additional value, which represents the number of steps to a counterexample. The `push()` procedure returns five different values. For clarity, we encapsulated these values in a structure called `PushResult`, as illustrated in Figure 5.7.

```
1  struct PushResult {
2      InductionFrame i_frame;
3      InductionFrame new_i_frame;
4      int n;
5      bool is_invalid;
6      int steps_to_ctx;
7      PushResult(InductionFrame i_frame,
8                 InductionFrame new_i_frame,
9                 int n,
10                bool is_invalid,
11                int steps_to_ctx) { ... }
12 };
```

Figure 5.7 PushResult structure

In the next step, we must update the `steps_to_ctx` value, which has been set to 0 during the initialization. This update occurs in the same part of the code where we set the `isInvalid` flag to `True`. The new value of `steps_to_ctx` will be the

sum of the number of steps required to reach the counterexample from `g_cex` (i.e., `g_cex.num_of_steps`) and a number `k` returned by the `CheckReachability()` function. This `k` tells us that `g_cex` is k-reachable from the initial states. Thus, the final value of `steps_to_ctx` will be `g_cex.num_of_steps + k`.

Finally, we need to correctly assign the previously used value `num_of_steps` for each newly created counterexample. We already analyzed in section 4.5 that a new counterexample is only generated once.. This process is displayed in a code snippet in Figure 5.8.

The new counterexample arises as a generalization of the previous potential counterexample, with its version incremented by `k`. Therefore, the value of `num_of_steps` will be the same as the `num_of_steps` of the previous potential counterexample, incremented by `k`.

```
1 PTRef f_cex = tm.sendFlaThroughTime(obligation.counter_example.ctx,
    i);
2 MainSolver solver2(logic, config, "f_cex reachability");
3 solver2.insertFormula(iframe_abs);
4 solver2.insertFormula(t_k_constr);
5 solver2.insertFormula(f_cex);
6 auto res2 = solver2.check();
7 if (res2 == s_True) {
8     auto model2 = solver2.getModel();
9     CounterExample g_cex(reachability_checker.generalize(*model2,
    t_k, f_cex), obligation.counter_example.num_of_steps + k);
```

Figure 5.8 CounterExample initialization

At last, we have to make a minor adjustment. In the pseudocode, the model is obtained directly from the `CheckSAT()` command, but in our implementation, we should only request the solver to provide the model if we have verified that the result of `CheckSAT()` was positive. Thus, we ask the solver for a model after an if statement in which we verify the satisfiability.

6 Experiments

In this chapter, we compare the performance of the PDKind engine with two Golem engines, SPACER and KIND.

6.1 Methodology

We measured the performance of each engine on the benchmarks from the CHC-COMP[1] in the year 2021. CHC-COMP is an annual competition that compares the performance of multiple CHC solvers.

We ran all three engines on all benchmarks from the LRA-TS category of CHC-COMP. This category focuses on transition systems over linear real arithmetic, with 498 benchmarks in it. In these experiments, the goal is to 1) verify the correctness of the PDKind engine by comparing results with the other two engines, and 2) find out how PDKind performs in comparison with the other two engines, where Spacer is the default engine for Golem, and KIND is the best-performing engine for LRA-TS in Golem.

6.2 Results

All experiments were conducted on a machine with an AMD EPYC 7452 32-core processor and 8 × 32 GiB of memory; the timeout was set to 300 s, where for each engine 8 processes were running in parallel. During the experiments, there were no conflicts in answers between the engines. We can consider this fact as strong proof of the correctness of our engine.

In the table 6.1, we can see that PDKind is significantly better than Spacer in the number of SAT results and matched it in the number of UNSAT results. KIND, however, outperformed both engines in both SAT and UNSAT results.

Result	PDKind	Spacer	KIND
SAT	242	214	260
UNSAT	68	70	84
TIMEOUT	188	214	154

Table 6.1 Number of solved benchmarks from LRA-TS category

In Figure 6.2 (The scale on both axes is logarithmical and represents the evaluation time in seconds), we compare the SAT results of the engines. In the first plot, we compare PDKind with Spacer. We can see that in most cases, PDKind is at least slightly faster than Spacer. In the second plot, where we compare PDKind with KIND, it is obvious that KIND outperforms our engine, but we can observe several cases where PDKind solved the problem in the given timespan of 300 seconds, but KIND timed out.

In Figure 6.3, we compare the UNSAT results of the engines. In both cases, our engine doesn't perform as well as the other engine, but in the first case, the

[1] `https://github.com/orgs/chc-comp/repositories`

performance of the PDKind engine is at least comparable with the Spacer engine, where the times are mostly similar, in some cases better for Spacer, in other better for PDKind.

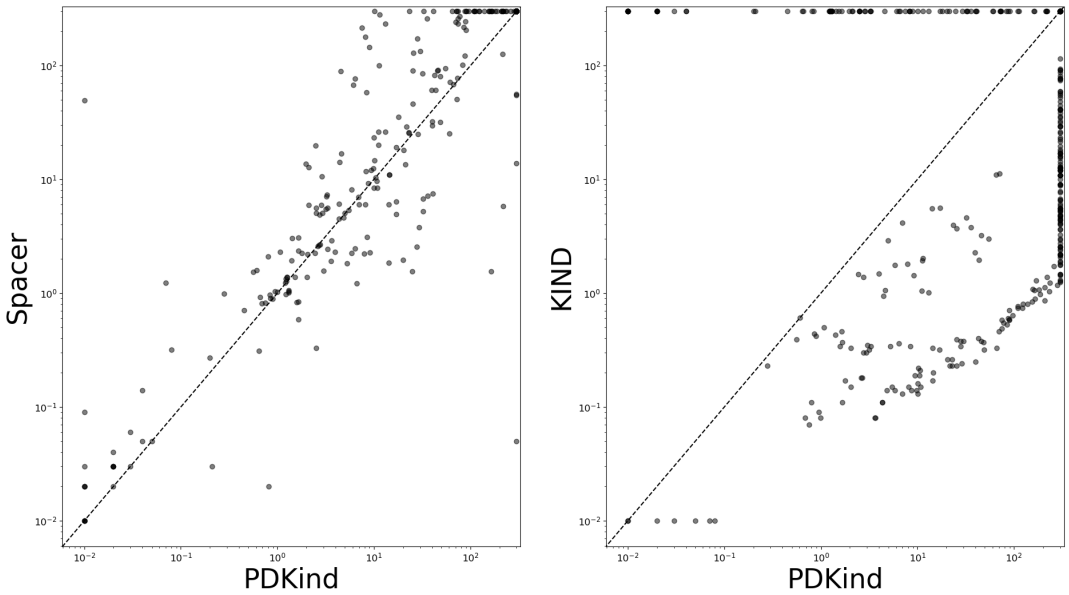

Figure 6.2 Time Comparison: SAT results

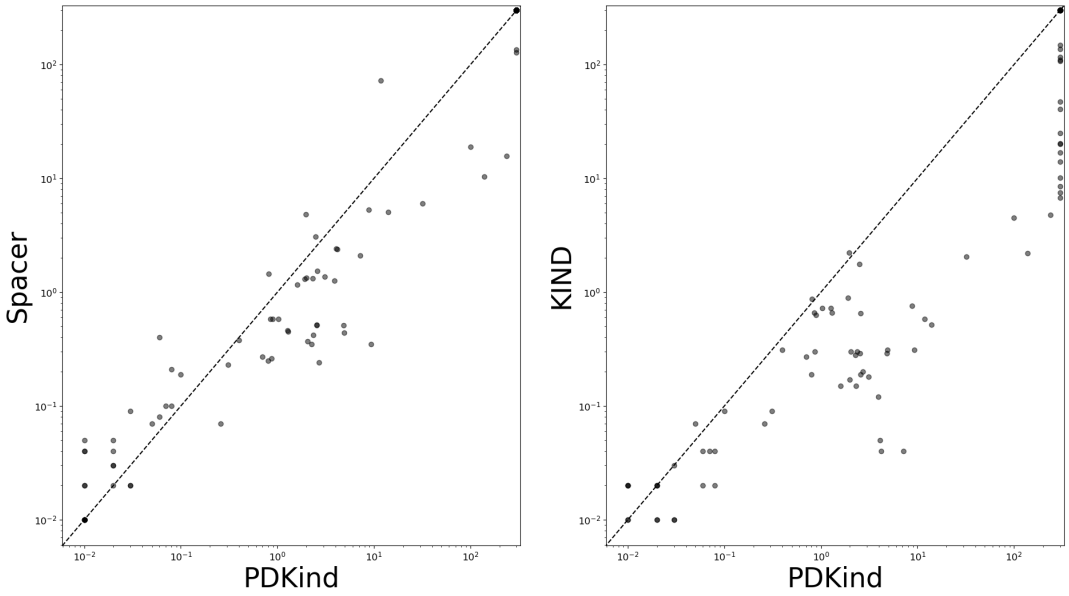

Figure 6.3 Time Comparison: UNSAT results

7 Conclusion

During this work, we introduced ourselves with the concept of the CHC framework. Then, we defined the concepts of transition systems and satisfiability modulo theories. After that, we described the structure of Golem and analyzed the best way to integrate a new engine into it. We then analyzed the entire PDKind algorithm and modified it to serve our needs.

With this knowledge, we implemented the PDKind engine itself. The goal was to create an engine that would 1) be well integrated into the Golem framework, 2) return correct answers, and 3) match the performance of other engines. To validate these goals, we experimentally compared our engine with other existing Golem engines. We managed to achieve results comparable to other engines, and in some cases, our engine was faster, although it did not reach the performance of the best-performing engine.

We believe that this engine presents a well-performing alternative to other engines, with its usefulness expected to grow with further improvements.

Bibliography

1. Barrett, Clark; Fontaine, Pascal; Tinelli, Cesare. *The Satisfiability Modulo Theories Library (SMT-LIB)* [www.SMT-LIB.org]. 2016.
2. Biere, Armin; Cimatti, Alessandro; Clarke, Edmund; Zhu, Yunshan. Symbolic Model Checking without BDDs. In: Cleaveland, W. Rance (ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. isbn 978-3-540-49059-3.
3. 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.
4. Bruttomesso, Roberto; Pek, Edgar; Sharygina, Natasha; Tsitovich, Aliaksei. The OpenSMT Solver. In: Esparza, Javier; Majumdar, Rupak (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 150–153. isbn 978-3-642-12002-2.
5. Cook, Stephen A. The complexity of theorem-proving procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. STOC '71. isbn 9781450374644. Available from doi: 10.1145/800157.805047.
6. Gurfinkel, Arie. Program Verification with Constrained Horn Clauses (Invited Paper). In: Shoham, Sharon; Vizel, Yakir (eds.). *Computer Aided Verification*. Cham: Springer International Publishing, 2022, pp. 19–29. isbn 978-3-031-13185-1.
7. 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.
8. 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.
9. Komuravelli, Anvesh; Gurfinkel, Arie; Chaki, Sagar. SMT-Based Model Checking for Recursive Programs. In: Biere, Armin; Bloem, Roderick (eds.). *Computer Aided Verification*. Cham: Springer International Publishing, 2014, pp. 17–34. isbn 978-3-319-08867-9.
10. McMillan, K. L. Interpolation and SAT-Based Model Checking. In: Hunt, Warren A.; Somenzi, Fabio (eds.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–13. isbn 978-3-540-45069-6.

11. McMillan, Kenneth L. Lazy Abstraction with Interpolants. In: Ball, Thomas; Jones, Robert B. (eds.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 123–136. isbn 978-3-540-37411-4.

12. Sheeran, Mary; Singh, Satnam; Stålmarck, Gunnar. Checking Safety Properties Using Induction and a SAT-Solver. In: Hunt, Warren A.; Johnson, Steven D. (eds.). *Formal Methods in Computer-Aided Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 127–144. isbn 978-3-540-40922-9.

13. Somenzi, Fabio; Bradley, Aaron R. IC3: Where monolithic and incremental meet. In: *2011 Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 3–8.

List of Figures

A Attachments

A.1 First Attachment