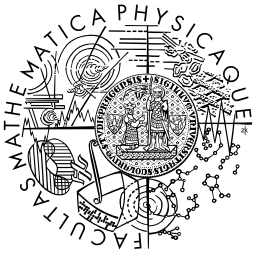

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Štěpán Henrych

The PD-KIND algorithm in the Golem CHC 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 CHC 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: PDKind (Property-Directed K-induction) is a combination of IC3 and k-induction, both commonly used model checking algorithms. PDKind separates reachability checking from induction reasoning, allowing induction to be replaced by k-induction. This work focuses on analysing and implementing the PDKind algorithm in the Golem solver as a back-end engine. By integrating PDKind into Golem, our goal is to take advantage of its unique approach to improve the solver's effectiveness in handling various verification tasks. The implementation will be tested and compared to other engines within Golem on a set of benchmarks to demonstrate its comparable efficiency.

Keywords: PDKind, Golem, Model checking

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: PDKind (Property-Directed K-induction) je kombinací IC3 a k-indukce, obou běžně používaných algoritmů pro model checking. PDKind odděluje ověření dosažitelnosti od indukčního odůvodňování, což umožňuje nahradit indukci k-indukcí. V této práci se zaměřujeme na analýzu a implementaci algoritmu PDKind jako back-endovém enginu v řešiči Golem. Integrací PDKind do Golemu chceme využít jeho jedinečný přístup ke zlepšení efektivity řešiče při řešení různých verifikačních úloh. Implementace bude testována a porovnána s ostatními enginy řešiče Golem na sadě benchmarků, aby se prokázala její srovnatelná účinnost.

Klíčová slova: PDKind, Golem, Model checking

Contents

1 Introduction

In computer science and software engineering, the idea of software verification is becoming increasingly important. Software verification is the process of checking whether a program or system operates correctly and meets its specification. The goal is to detect bugs and errors during the development process and ensure reliability, especially in critical systems.

There are several approaches to software verification. Some are based on interactive theorem proving, where the user manually constructs proofs using logical frameworks. Others rely on static analysis techniques that over-approximate a program's behavior to detect potential bugs without executing the code.

In contrast, **model checking** is a fully automated technique that explores the state space of a system and checks whether the desired properties hold. It is especially suited for verifying safety properties, which ensure that "nothing bad ever happens" during execution. These properties are typically expressed as logical formulas over program states, and the system is modeled as a **transition system**, which defines how the system evolves from one state to another.

One formalism gaining popularity in this area is the Constrained Horn Clauses (CHC)[9] framework. CHCs form a fragment of first-order logic modulo constraints that can express many program verification problems as constraint-solving tasks. The key advantage of CHCs is that they separate modeling from solving by translating the program's behavior and properties into logical clauses, which can then be passed to a solver to decide satisfiability.

Golem[5] is one such solver that integrates the interpolating SMT solver OpenSMT[11]. Golem currently implements six model-checking algorithms for solving CHC satisfiability problems.

In addition to solving the satisfiability problem, each engine in Golem provides a validity witness for its answer. In software verification, these witnesses can be viewed as inductive invariants for SAFE answers and counterexample paths for UNSAFE answers. By including a witness, the engine's answer can be validated. Moreover, to guard against incorrect witnesses, Golem has a built-in internal validator that checks each witness's correctness.

One such algorithm for solving CHC problems is PDKind (Property-Directed K-induction)[12]. PDKind is an IC3[18]-based algorithm that separates reachability checking from induction reasoning, allowing the induction step to be replaced with k-induction.

While IC3 uses simple induction, which may not be sufficient to prove more complex properties, k-induction can consider multiple steps at once, making it more powerful in some cases. By combining the structure of IC3 with the strength of k-induction, PDKind aims to improve the performance of model checking, especially when verifying difficult safety properties.

1.1 Goals

The goal of this work is to implement the PDKind algorithm as a new engine and integrate it into the Golem solver[5].

First, we present the necessary background on SMT solving, model checking,

and CHCs. Then, we introduce the architecture of the Golem solver and analyze the PDKind algorithm in detail. Based on this analysis, we design a solution for integrating PDKind as a new engine in Golem.

Next, we implement the design and extend it to generate validity witnesses for both SAFE and UNSAFE answers. We evaluate the implementation on a set of benchmarks, comparing its correctness and efficiency with the other existing engines in Golem.

Finally, we use Golem's internal validator to verify that the produced witnesses are correct. Throughout the process, we reflect on the design decisions and implementation challenges, with the aim of contributing a reliable and reusable component to the Golem verification ecosystem.

2 Background

2.1 Transition System

Transition systems provide a framework for modeling how systems change over time through state transitions. They are extensively used in formal methods, program analysis, and verification. Transition systems consist of a set of initial states, a set of states, and a transition relation that defines the evolution of states. Transition systems are used as an abstraction in computer science and software verification to solve several problems, including the Constrained Horn Clause's (CHC) satisfiability. By offering a formal method of representing the behavior of a computer system or an abstract computing device, they make it easier to analyze and verify properties. We will introduce the basic ideas of transition systems in this chapter, which will help us understand the following chapters.

Definition 1. **Transition System [12]:** A transition system is a 4-tuple (S, $\mathcal{F}$, $\mathcal{T}$, $\mathcal{I}$), where:

- S is a finite set of state variables $\vec{x}$. Each $x \in \vec{x}$ has a primed version x' representing its value in the next state. A **state** s is a valuation over $\vec{x}$ that assigns a value $s(x)$ to each $x \in \vec{x}$.
- $\mathcal{F}$ is a set of quantifier-free formulas, where each $F(\vec{x}) \in \mathcal{F}$ is a **state formula** that holds in a state s if $s \models F$.
- $\mathcal{T}$ is a set of quantifier-free formulas, where each $T(\vec{x}, \vec{x}') \in \mathcal{T}$ is a **transition formula** describing valid transitions between states. A transition from state s to s' is allowed if $(s, s') \models T$ for some $T \in \mathcal{T}$.
- $\mathcal{I}$ is a set of formulas, where each $I(\vec{x}) \in \mathcal{I}$ is a **state formula** describing initial states. A state s is an initial state if $s \models I$.

Definition 2. **Successor [12]:** State s has a successor s', when $T(s(\vec{x}), s'(\vec{x}'))$ is true.

Definition 3. **k-Reachability [12]:** 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.

2.2 Safety of Transition System

An essential concept in the analysis of transition systems is safety. We consider a transition system to be safe if it never reaches an unsafe state during its execution, where an unsafe state is one in which the desired property is violated. This notion of safety is crucial for verifying the correctness of programs and systems modeled by transition systems.

Definition 4. **Safety Property:** A property P is called a **safety property** if it ensures that "nothing bad will ever happen" during the execution of the system. Formally, given a transition system $(S, \mathcal{F}, T, I)$ and a property P (a state formula that can be evaluated on each state), we say P is a safety property if for every reachable state $s \in S$, $P(s)$ is true. Equivalently, the system never reaches a state where $P(s)$ is false (such a state is considered unsafe).

Definition 5. **Invariant [12]:** An **invariant** is a property that holds in all reachable states of a transition system. Formally, for a transition system $(S, \mathcal{F}, T, I)$, a state formula I is an invariant if every reachable state $s \in S$ satisfies $I(s)$.

Safety properties can often be expressed as invariants, since they require that the system never enters an unsafe state.

Verifying Safety. To verify whether a transition system satisfies a given safety property P, we must check that all reachable states of the system satisfy P. There are several approaches to perform this verification:

- **Explicit State Exploration.**

  This technique enumerates all reachable states of the system and checks whether the safety property holds in each of them. While simple and complete for finite systems, it quickly becomes infeasible due to the state explosion problem. Tools like SPIN[10] apply optimizations to mitigate this.

- **Bounded Model Checking (BMC).**

  BMC checks whether a counterexample of length k exists by unrolling the transition system k steps and using a SAT/SMT solver to find violations. If no counterexample is found, the property holds up to depth k, but this alone is not enough to prove the property.

  To prove unbounded safety, BMC is often combined with k-induction7. This method verifies that the property holds for all states reachable within k steps and that if the property holds for k steps, it holds for the $k+1$-th as well. If both checks succeed, the property is proven for all executions.

- **Inductive Invariant Checking.** This method tries to find an invariant $Inv(x)$ that holds initially, is preserved by transitions, and implies the safety property.

  If such an invariant is found, it proves the property. Algorithms like IC3[18] automatically construct inductive invariants. Many verification tools encode these problems as Constrained Horn Clauses (CHCs) and solve them using SMT-based solvers.

Example: Consider a transition system modeling a simple counter with state variable x that starts at 0 and is incremented by 1 at each step. Formally, let the initial condition be $\mathcal{I}$ and the transition relation be $\mathcal{T}$.

$$\mathcal{I} : x = 0$$
$$\mathcal{T} : x' = x + 1$$

Now suppose we have a safety property P stating that "the counter never exceeds 10," i.e., $P(s)$ is $(x \leq 10)$ for a state s. In this system, P is actually violated after few steps: for instance, after 11 transitions, the counter reaches a state where $x = 11$, which is an unsafe state since it does not satisfy P. We can illustrate how each verification technique would handle this scenario.

An explicit state exploration would enumerate states $x = 0, 1, 2, \ldots$ and eventually encounter $x = 11$, identifying it as a counterexample that violates the safety property.

Using BMC, if we set the bound $k = 10$, the model checker will not find any violation since no state with $x > 10$ is reachable within 10 steps. Increasing the bound to $k = 11$ will produce a counterexample trace $s_0 \to s_1 \to \cdots \to s_{11}$ where s_0 has $x = 0$ and s_{11} has $x = 11$, demonstrating that an unsafe state is reachable. This shows that BMC can detect the error when the bound is sufficiently large, but it cannot conclude overall safety without considering an unbounded number of steps.

With inductive invariant checking, we would attempt to prove the property $x \leq 10$ by induction. In this case, no suitable inductive invariant exists to prove $x \leq 10$ for all reachable states because the property itself is false for reachable state $x = 11$. The inductive step fails. As a result, the method reports that it cannot prove the property, effectively uncovering the same counterexample at the point where the induction fails.

Ensuring safety is critical for system reliability. By using the appropriate verification technique (or a combination of techniques), we can determine whether a system satisfies its safety properties or find counterexamples that demonstrate violations.

2.3 Satisfiability Modulo Theories[2]

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 [7]. 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, including real numbers, integers, or even lists and other data structures. In other words, an SMT instance arises as a generalization of a SAT instance, where Boolean variables are replaced with predicates of some theory. Examples of such theories are uninterpreted functions (UF), arrays, linear arithmetic (LA), or, more specifically, linear real arithmetic (LRA), which we will focus on in the rest of this work.

SMT solvers are widely used in various fields, such as formal verification, model checking, program synthesis, and hardware verification. They are instrumental in solving real-world problems that involve reasoning over complex theories, such as reasoning about hardware designs, program correctness, and even automated theorem proving.

The concept of SMT was introduced in the late 1990s as a way to extend SAT to handle more expressive logical formulas. SMT solvers combine techniques from both SAT solving and decision procedures for various theories, making them more powerful and versatile than traditional SAT solvers.

Each theory in SMT corresponds to a set of logical rules and operations for a specific type of data. For instance, in the case of Linear Real Arithmetic (LRA), the theory defines operations and constraints over real numbers. SMT solvers

then check the satisfiability of a formula under a combination of such theories, making it more expressive than just Boolean logic.

For example, consider a system where we need to check if there is an assignment of integer values x and y that satisfies the following conditions:

$$2x + y \geq 10 \wedge x - y = 3$$

A traditional SAT solver would not be able to handle this problem, as it involves arithmetic over integers. However, using the linear arithmetic (LA) theory, an SMT solver can quickly solve this problem.

To find a satisfying assignment, we solve the system:

$$\begin{aligned} x - y &= 3 \\ 2x + y &\geq 10 \end{aligned}$$

From the first equation, we isolate x:

$$x = y + 3$$

Substitute into the second inequality:

$$2(y + 3) + y \geq 10 \Rightarrow 2y + 6 + y \geq 10 \Rightarrow 3y \geq 4 \Rightarrow y \geq \frac{4}{3}$$

Then

$$x = y + 3 \Rightarrow x \geq \frac{4}{3} + 3 = \frac{13}{3}$$

So any pair (x, y) such that $y \geq \frac{4}{3}$ and $x = y + 3$ is a solution. For example:

$$y = 2, \quad x = 5$$

is a valid model:

$$2 \cdot 5 + 2 = 12 \geq 10, \quad 5 - 2 = 3$$

While SAT solvers are only used for assigning values for Boolean variables, SMT solvers work with more complex theories, allowing them to handle constraints over data types beyond just true or false values. This extension makes SMT a powerful tool for solving problems that SAT solvers cannot address.

2.4 CHC Satisfiability

Constrained Horn Clauses (CHCs)[9] is a fragment of First Order Logic modulo constraints that capture many program verification problems as constraint solving.

They extend Horn Clauses, which are logical formulas of the form:

$$P_1 \wedge P_2 \wedge \cdots \wedge P_n \implies H$$

where P_i are body literals and H is the head of the clause.

In CHCs, the literals in the body can include constraints in a first-order theory, such as linear arithmetics, boolean logic, etc. Therefore, CHC is a formula of the form:

$$\phi \wedge P_1 \wedge P_2 \wedge \cdots \wedge P_n \implies H$$

where ϕ is a constraint in the first order theory, P_i and H are uninterpreted predicates.

A CHC is satisfiable if there exists an interpretation of the uninterpreted predciates P_i and H that is valid in the background theory.

The main advantage of CHC is that it separates the 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.

For example, we consider a program with a loop:

```
int i → 10;

while (x > 0) {
    x → x - 1;
}

assert (x >= 0);
```

We can encode the behavior and safety of this program in CHCs:

Initial clause:

$$x = 10 \implies P(x) \tag{2.1}$$

Transition clause:

$$P(x) \wedge x > 0 \wedge x' = x - 1 \implies P(x') \tag{2.2}$$

Safety clause:

$$P(x) \wedge x < 0 \implies False \tag{2.3}$$

In this simple example, we encode the program using three CHC clauses. (2.1) describes the initial state of the program and introduces an uninterpreted predicate $P(x)$, which represents the loop invariant. (2.2) describes one step of the loop's transition: it checks if $x > 0$ (the loop condition) and $P(x)$ (the invariant), then decrements x. Satisfiability of the left-hand side of this clause implies that the invariant $P(x')$ holds for the decremented value of x. Finally, (2.3) checks the assertion condition. We verify the assertion by checking if its negation leads to a contradiction. If the negation is satisfiable, it implies a false state, indicating that the assertion does not hold.

In the previous example, we demonstrated how to encode a simple loop using Constrained Horn Clauses (CHCs). This encoding process is not limited to simple loops and can be extended to more complex and less deterministic loops. By representing the loop's initial state, transition, and termination conditions as CHCs, we can analyze the program's behavior over a series of execution steps. We can introduce a compact representation of a loop in CHCs as follows.

$$\begin{aligned} Init(V) &\implies Inv(V) \\ Inv(V) \wedge Tr(V, V') &\implies Inv(V') \\ Inv(V) \wedge Bad(V) &\implies False \end{aligned} \tag{2.4}$$

This representation allows us to analyze the loop's behavior over a finite number of iterations.

2.5 OpenSMT

Golem uses OpenSMT[6] as its underlying SMT solver to handle formulas that arise during CHC solving. OpenSMT supports several background theories and provides features like interpolation, which are necessary for some of the algorithms implemented in Golem.

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[2] 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[11], which supports interpolation for propositional logic, QF_UF, QF_LRA, QF_LIA, and QF_IDL.

2.6 Induction vs k-Induction

The PDKind algorithm is a combination of IC3 and k-induction. IC3[18] is a commonly used method that uses induction to show a property is invariant by incrementally constructing an inductive strengthening of the property. PDKind [12] breaks IC3 into modules, which allows replacing the induction method with k-induction.

We need to introduce two definitions to compare the relative strength of induction and k-induction. Let us have a transition system $(S, \mathcal{F}, \mathcal{T}, \mathcal{I})$.

Definition 6. ($\mathcal{F}$-Induction[12]): Let $\mathcal{F}$ be a set of state formulas, where $P \in \mathcal{F}$. Then P is $\mathcal{F}$-inductive if:

$$I(\vec{x}) \implies \mathcal{F}(\vec{x}), \tag{init}$$
$$\mathcal{F}(\vec{x}) \wedge T(\vec{x}, \vec{x}') \implies P(\vec{x}'). \tag{cons}$$

If $\mathcal{F} = P$, we say that P is inductive,

[12] states that if P is inductive, it is also invariant. However, invariants are generally not inductive. To prove that P is an invariant, we need to find a set

of formulas $\mathcal{F}$ that includes P and is itself inductive. This set is also called the strengthening of P. If such a strengthening exists, then P must be an invariant.

Definition 7. ($\mathcal{F}^k$-Induction[12]):Let $\mathcal{F}$ be a set of state formulas, where $P \in \mathcal{F}$. Then P is $\mathcal{F}^k$-inductive if:

$$I(\vec{x_0}) \wedge \bigwedge_{i=0}^{l-1} T(\vec{x_i}, \vec{x}_{i+1}) \implies \mathcal{F}(\vec{x_l}), \qquad \text{for } 0 \leq l < k \quad \text{(k-init)}$$

$$\bigwedge_{i=0}^{k-1} (\mathcal{F}(\vec{x_i}) \wedge T(\vec{x_i}, \vec{x}_{i+1})) \implies P(\vec{x_k}). \qquad \text{(k-cons)}$$

If $\mathcal{F} = P$, we say that P is k-inductive.

If we substitute k with 1 in k-induction definition7, we can see that a property that is inductive is 1-inductive and also k-inductive for any k. In the other direction, [12] shows that if a property P is k-inductive and the logical theory allows quantifier elimination, then an inductive strengthening of P can be constructed by interpolation.

For theories that admit quantifier elimination, k-induction and induction have the same deductive power. However, [4] shows that k-induction can yield more succinct strengthening. For other theories, such as Boolean logic or Linear arithmetic, k-induction may be exponentially more powerful than induction due to weaker interpolation.

Practical Effectiveness

[12] shows that k-Induction is effective, especially when combined with algorithms like IC3. [12] demonstrates the effectiveness of k-induction on an example. The same example is depicted in Figure 2.1.

```
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 2.1 Induction vs K-induction example

The example shows a transition system, where zeroes are written to an array in a circular manner. The goal is to prove that the property $a[0] == 0$ is true throughout the execution of the program.

Induction:

To prove P using induction, we must show the following:

- **Initial state**: P holds in the initial state. This is true since $a[0] = 0$
- **Inductive step**: If P holds in state k, it must hold in state $k + 1$

Here induction fails because it only considers a single step, which doesn't guarantee that $a[j]$ is always defined correctly. Thus, the program could copy a value from an undefined location to $a[0]$, which could break the property.

K-Induction:

While induction attempts to prove P in a single step, k-induction considers a sequence of steps. In this case, choosing $k = N + 1$ allows the index i to cycle back to 0, covering a complete cycle of transitions.

Over these $N+1$ steps, every position in the array copies a zero from a previous index, eventually filling the entire array with zeros. This guarantees that $a[0] = 0$ always holds. Thus, P is $(N + 1)$-inductive.

This example illustrates how k-induction succeeds where simple induction fails, particularly in systems with state transitions that unfold over multiple steps.

3 Golem

Golem [5] 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[11], 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

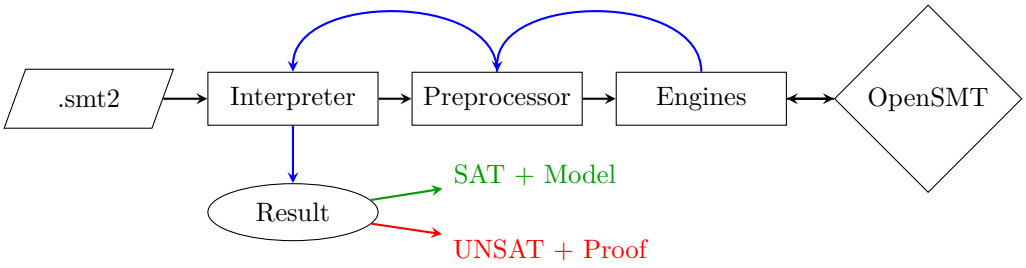

Figure 3.1 Architecture of Golem

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[2]. 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) [3]
- k-Induction (KIND) [17]
- Interpolation-based Model Checking (IMC) [14]
- Lazy Abstractions with Interpolants (LAWI) [15]
- Spacer [13]
- Transition Power Abstraction (TPA) [5]

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.

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

Figure 3.2 UNSAT + Proof example

In Figure 3.2, we can see a CHC system and a proof of its unsatisfiability. There are four derivation steps. In the first step, the unsatisfiability proof shows that the variable x is set to 1, giving us $I(1) \impliedby True$. In the second step, we continue with the initial value of x and proceed to get $x' := 2$ which gives us $T(1, 2) \impliedby True$. Step three applies resolution to the instance of the third clause for $x := 1$ and $x' := 2$ and the previously derived facts $I(1)$ and $T(1, 2)$, giving us $S(2) \impliedby True$. The last step again applies resolution to the instance of the fourth clause with $x := 2$ and the derived fact $S(2)$ resulting in $False$ clause.

Therefore, the proof gives us the complete trace from initial states resulting with a false clause.

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 an interesting option, as it would allow us to leverage certain advantages of other programming languages and make our engine compatible with other solvers and verification programs. However, integrating C++ with another language may introduce

some performance overhead. This overhead could be caused by complex data conversions, memory allocation issues, or additional context switching. The extent of this overhead depends on how we plan to utilize the library. If the library was a standalone engine that received input from Golem and returned an SAT result, the overhead would be minimal. However, if we wanted the library to interact with Golem more, such as by using its SMT solver or storing additional information about the solving process, the performance could degrade.

The goal of this work is to create a new efficient engine within the Golem solver and compare its performance with other Golem engines. As we said above, using or implementing a different SMT solver and isolating the solving process in a library would not bring any noticeable overhead. Therefore it could look like a feasible solution. However, choosing this approach would undermine the purpose of this work. We would not be able to compare the performance because the comparison of different engines would be affected by the performance of their underlying SMT solvers.

Another option would be to write a C++ library and include it in the project. The big advantage of this solution would be the option of using the engine library in other solvers. Using C++ would also eliminate the mentioned downsides of the first option, but it would still leave some. For example, to keep the engine universal, we must integrate an SMT solver for satisfiability checking and interpolation. That, as we have already discussed, would not be an optimal solution for the purpose of this work.

The last option would be to utilize Golem for SMT solving. With this approach, we would use C++ to achieve high performance, and we would use Golem's integrated SMT solver to eliminate the above-mentioned disadvantages such as not being able to effectively compare engines with different SMT solvers. On the other hand, such an approach would make our engine usable only for the Golem solver, and it could not be used as a standalone engine in other solvers.

The most suitable option for achieving our goal is to integrate the engine directly into the Golem solver, following the approach used for its existing engines. This approach allows us to fully leverage OpenSMT, Golem's integrated SMT solver, for satisfiability checking and interpolation. Additionally, we can take advantage of other Golem's built-in features, which would simplify development and improve efficiency. Although this method ties the engine to Golem, it is ideal for the work's goal of developing a high-performance engine for Golem and comparing it against its other engines.

4 PDKind

PDKind is an IC3[18]-based algorithm, that separates reachability checking and induction reasoning, allowing the induction core to be replaced with k-induction.

In this chapter, we describe and analyze the PDKind algorithm. In sections 4.1-4.3, we will be using methodology, algorithms, and definitions described in [12]. After reading this chapter, the reader should be able to understand the parts of the PDKind algorithm and orientate in our implementation.

4.1 PDKind Main Procedure

The PDKind algorithm aims to prove a safety property P is invariant or to find a counterexample. It achieves this by separating the process into two subprocedures, which work together within the main loop. Before diving into the algorithm, we introduce the key data structures and subprocedures it uses. *Reachability frames* for the reachability checking, and *Induction frames* for managing induction obligations.

Reachability frames $R_0, R_1, \ldots, R_k$ are used to guide the reachability checking 4.2. Each frame R_i is an over-approximation of the set of states from which the target formula is reachable in exactly $k - i$ steps. In particular, R_k corresponds to the set of states satisfying the target formula (the "bad" states), and R_0 represents states that can reach the target in k steps. These reachability frames help avoid redundant work by caching which portions of the state space have already been shown to be unreachable, thereby preventing the algorithm from re-exploring those paths.

Induction frames, on the other hand, are used by the Push subprocedure in section 4.3 to store and refine candidate invariants. An induction frame F at a given depth n is essentially a set of pairs $(lemma, counterExample)$, where each $lemma$ is a formula (state property) that is supposed to hold as an invariant up to depth n, and $counterExample$ is a description of states that $lemma$ aims to exclude at depth n. Initially, the induction frame is initialized with the pair $(P, \neg P)$, meaning our starting candidate invariant is P itself, and the "counterexample" to this invariant is $\neg P$ (the property's negation). This indicates that until we strengthen P, the primary bad state to consider is any state satisfying $\neg P$.

Using these structures, the main PDKind procedure coordinates the model-checking process. In each iteration, it calls the reachability checking and push subprocedures to either extend the proof of P or find a real counterexample. Algorithm 4.1 outlines this top-level procedure. The main loop works as follows: we maintain an induction frame F (starting with $(P, \neg P)$) at depth n (starting at $n = 0$). At each step, we pick a k (typically $k = n + 1$) and invoke the Push procedure on the current frame F to attempt to find a k-inductive strengthening of P. In other words, Push will try to either prove that F (or an extension of F) is inductive for k steps, or identify how F fails and strengthen it by adding new lemmas.

The Push procedure 4.3 internally uses the reachability checking procedure to check whether certain potential counterexample states are actually reachable

from the initial state. If the Push procedure discovers a concrete counterexample trace from an initial state to a bad state, it will mark the property as invalid. Otherwise, Push returns a strengthened set of lemmas (a new induction frame G) that is valid up to a higher depth.

After each call to Push, the main algorithm examines the results:

- If Push reported that the property is invalid (i.e., a counterexample was found), then the main procedure terminates and returns UNSAFE, as we have proof that P is not an invariant.
- If the new frame G returned by Push is identical to the old frame F (i.e. $G = F$, meaning Push did not add any new lemmas), then F is already k-inductive. In this case, no further strengthening is required—P has been proven invariant—so the algorithm returns SAFE.
- Otherwise (the property is not yet proven, but no counterexample was found), the algorithm increases the induction depth and continues. We update n to the new value n_p provided by Push (which is at least $n + 1$), set $F := G$ (using the strengthened frame as the new current frame), and repeat the loop for the next iteration.

In this way, the PDKind main procedure iteratively strengthens the candidate invariant P with new lemmas and deepens the search, until either a valid inductive invariant is achieved or a real counterexample is discovered.

```
1: Input: Initial states I, transition formula T, property P
2: Output: Return UNSAFE if P is invalid or SAFE when there is no
   inductive strengthening left
3: n ← 0
4: F ← (P, ¬P)
5: while true do
6:     k ← n + 1
7:     (F, G, n_p, isInvalid) ← 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 ← n_p
15:    F ← G
16: end while
```

Figure 4.1 Main PDKind procedure

4.2 Reachability checking procedure

To understand the following, we need to introduce a few definitions.

Definition 8. Given a transition T and a state formula F. For $k > 1$, $T[F]^k(\vec{x}, \vec{x}')$ is defined as

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

This represents the unrolling of the transition relation T over k steps, where F holds in the intermediate states $\vec{w}$.

Not only in this section, but in the whole algorithm we will be using two important techniques, interpolation and generalization. Let us have a formula in the form

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

where $T[B]^k(\vec{x}, \vec{w}, \vec{y})$ follows the previous definition 8, and $\vec{x}$, $\vec{w}$, and $\vec{y}$ represent the initial, intermediate, and final state variables, respectively.

Definition 9. **(Interpolant)** [12] 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.

Interpolants approximate the set of reachable states from a source $A(\vec{x})$ and exclude a target $C(\vec{y})$. They are used when the formula is unsatisfiable and help rule out unreachable parts of the state space by over-approximating the reachable states.

Definition 10. **(Generalization)** [12] 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.

When the formula is satisfiable, we generalize the final state to an under-approximation that still satisfies the formula but covers more states. This helps us work with a broader set of counterexamples and makes the recursive check more efficient.

```
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

The core part of PDKind is a depth-first reachability procedure that checks whether the target formula is reachable in exactly k steps. Instead of searching forward from the initial states, it works backwards from the target and tries to find a trace of the required length that can be extended to the initial states.

To make the search more efficient, the algorithm maintains reachability frames $R_0, \ldots, R_k$, where each frame R_i over-approximates the set of states that can reach the target in exactly $k-i$ steps. These frames help avoid repeating work and rule out parts of the state space that have already been shown to be unreachable.

When a satisfying trace is found, the final state is generalized into an under-approximation that still satisfies the formula but includes more states. The recursive check is then applied to this generalization. If the trace can't be extended to the initial states, the algorithm learns an interpolant that blocks it and adds it to the corresponding frame to avoid exploring similar paths again.

This procedure is used repeatedly in PDKind and forms the main loop of the algorithm. The pseudocode 4.2 describes it in more detail.

4.3 Push procedure

We have already metioned that the Push procedure utilizes data structure called Induction frame. Here we formally define it.

Definition 11. **(Induction Frame)** [12] 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 blocks the states described by *counterExample*, preventing them from occurring at depth n
- *counterExample* states can be extended to a counterexample to P.

In essence, an induction frame tracks the progressive strengthening of an invariant over multiple iterations. Each tuple in F consists of a candidate invariant formula (the lemma) and a specific "bad" state (the counterexample) that the lemma is meant to eliminate from consideration at the current depth. As the algorithm runs, these frames are updated: formulas are refined to rule out newly discovered bad states, strengthening the invariant.

Induction frame tracks the strengthening of invariants over number of iterations. Every frame consists of a formula that has been refined to eliminate an invalid state.

The Push procedure 4.3 is the core of the PDKind algorithm's inductive reasoning stage. It takes as input the current induction frame F at index n (which contains the current set of lemmas and their associated counterexamples) and attempts to push these lemmas to a higher inductive step (specifically, up to $k = n+1$) by analyzing counterexamples. In other words, Push tries to either prove each lemma holds one step further or to find why it fails and strengthen the frame accordingly. Throughout this process, Push alternates between generalization and reachability checks. We now break down the Push algorithm step by step.

(1) Initial k-inductiveness check:
For each obligation $(lemma, counterExample)$ in the frame F, Push first checks whether $lemma$ is k-inductive (see Definition 7). This is done by querying the SMT solver to check the satisfiability of $F_{ABS} \wedge T^k \wedge \check{n} lemma$, where F_{ABS} denotes the conjunction of all lemmas in F and T^k represents k steps of the transition relation. If this formula is unsatisfiable, it means that $lemma$ is already strong enough up to $n + k$ steps, and we can simply carry this obligation into the new induction frame G and continue to the next obligation. If instead the formula is satisfiable, the solver produces a model m_1 witnessing the failure of k-inductiveness (meaning m_1 is a state at depth k where $lemma$ does not hold under the current assumptions). We save this counterexample state m_1 for later analysis and do not yet add $lemma$ to G.

(2) Counterexample reachability check:
Next, the Push procedure examines the current *counterExample* associated with the lemma. It checks whether this *counterExample* could actually occur in a concrete execution. This is done by checking satisfiability of $F_{ABS} \wedge T^k \wedge$ *counterExample*, which asks whether there exists a trace of length k consistent with our current abstraction F_{ABS} that ends in a state satisfying *counterExample*. If this check returns satisfiable, we obtain a witness model m_2 that demonstrates a state at depth k matching the counterexample. We then generalize m_2 to a formula g_2. At this point, we know that from any state described by g_2, we can reach a violation of P (because g_2 over-approximates the counterexample at depth k, which leads to $\neg P$ at depth k). Now we must check if any initial state can lead to a state in g_2 within k steps. We call $\textsc{Reachable}(i, j, g_2)$ with to see if g_2 lies on some path from the initial states of length between i and j (essentially up to n steps). If this reachability check returns true, then we have discovered a concrete

counterexample execution from an initial state to a state that leads to $\neg P$. In this case, the property P is truly violated; we set $isInvalid := \text{true}$, indicating that a counterexample has been found, and we can break out of the Push loop early. If, on the other hand, the reachability procedure determines that g_2 is not reachable from any initial state, then m_2 was a spurious counterexample under the current abstraction. The reachability check will also provide an interpolant i_1 in this scenario, which is a formula that separates the initial states from the generalized counterexample g_2. Specifically, i_1 blocks all paths to g_2. We use this i_1 to refine our invariant by strengthening the current lemma. We then add the new obligation (i_1, g_2) to the current frame F. By doing this, we have eliminated the potential counterexample g_2, and we "try again" with a strengthened invariant.

(3) Induction failure analysis:
After handling the $counterExample$, the Push procedure returns to address the initial inductiveness failure captured by m_1 from step (1). Now we consider the generalized form of m_1, which is $g_1 = \textsc{Generalize}(m_1, T^k, \neg lemma)$, which represents a set of states at depth k where $lemma$ could fail. We need to check whether this scenario is genuine or not. Using the reachability check again, we ask if g_1 is reachable from the initial state. If g_1 is reachable from some initial state, then the failure of $lemma$ at depth k corresponds to a real counterexample scenario. We take the original $counterExample$ and add a new obligation $(\neg counterExample, counterExample)$ to both the current frame F and the new frame G. This new lemma $\neg counterExample$ is a weaker condition which is trivially true up to depth n but will block the counterexample at depth $n + 1$. By pushing $(\neg counterExample, counterExample)$ into F and G, we ensure that this particular bad state is explicitly ruled out in the future, and we remove the original stronger (but non-inductive) lemma from consideration.

If g_1 is not reachable from any initial state, we can salvage the lemma by strengthening it. We compute g_3 (an interpolant-based strengthening) by combining i_2 with the original $lemma$ to form a new lema that avoids the counterexample. We replace the old $(lemma, counterExample)$ in F with $(g_3 \wedge lemma, counterExample)$, and push this new tuple back into F for further processing. This way, the lemma has been fixed to be true up to depth $n + 1$.

After these steps, the Push procedure continues to process any remaining obligations in F (each of which goes through the same sequence of checks and potential updates). If at any point a real counterexample was found ($isInvalid = \text{true}$), Push will terminate early and propagate that information to the main procedure. Otherwise, once all obligations are handled, Push produces a new induction frame G containing all the lemmas that have been confirmed or strengthened for the next depth. It also provides an updated index n_p the new depth up to which G's lemmas are valid. The original frame F may also have been updated with strengthened lemmas during this process. Finally, Push returns $(F, G, n_p, isInvalid)$ to the main PDKind procedure. The main loop 4.1 will then use these results to decide whether to declare the property proven (if $F = G$), to declare it refuted (if $isInvalid$ is true), or to continue the process with $F := G$ and $n := n_p$ for another iteration.

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: $F \leftarrow F \cup (i_1, g_2)$
25: Q.push$((i_1, g_2))$
26: Q.push$(lemma, counterExample)$
27: Continue
28: **end if**
29: **end if**
30: $g_1 \leftarrow \text{Generalize}(m_1,\ T_k,\ \neg lemma)$
31: $(r_2, i_2, n_2) \leftarrow \text{Reachable}(n - k + 1,\ n,\ g_1)$
32: **if** r_2 **then**
33: $(r_3, n_3) \leftarrow \text{Reachable}(n + 1,\ n_2 + k,\ g_1)$
34: $n_p \leftarrow \text{Min}(n_p,\ n_3)$
35: $F \leftarrow F \cup (\neg counterExample,\ counterExample)$
36: $G \leftarrow G \cup (\neg counterExample,\ counterExample)$
37: **else**
38: $g_3 \leftarrow i_2 \wedge lemma$
39: $F \leftarrow F \cup (g_3,\ counterExample)$
40: $F \leftarrow F \setminus (lemma,\ counterExample)$
41: Q.push$((g_3,\ counterExample))$
42: **end if**
43: **return** $(F,\ G,\ n_p,\ isInvalid)$
44: **end while**

Algorithm 4.3 Push procedure

5 Implementation

In this chapter, we will analyze the PDKind algorithm and describe how it was adapted to our implementation. We will go into the structure of the implementation, cover the individual components of the engine, and explain the decisions made during development. Finally, we will describe how the engine was integrated into the Golem solver.

To keep the structure consistent with Golem's engine system, we placed our solution in the files `PDKind.cpp` and `PDKind.h`, without creating separate files for individual components.

5.1 Golem structures

In this section, we will describe the Golem's data structures and procedures that we utilize in our solution.

5.1.1 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`.

5.1.2 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.

Otherwise, 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.

5.2 PDKind structures

In this section, we analyze and describe the data structures and procedures implemented in our engine satisfying the methods and definitions shown in chapter 4. Also we describe how they interact with the OpenSMT solver, as shown in figure 5.3.

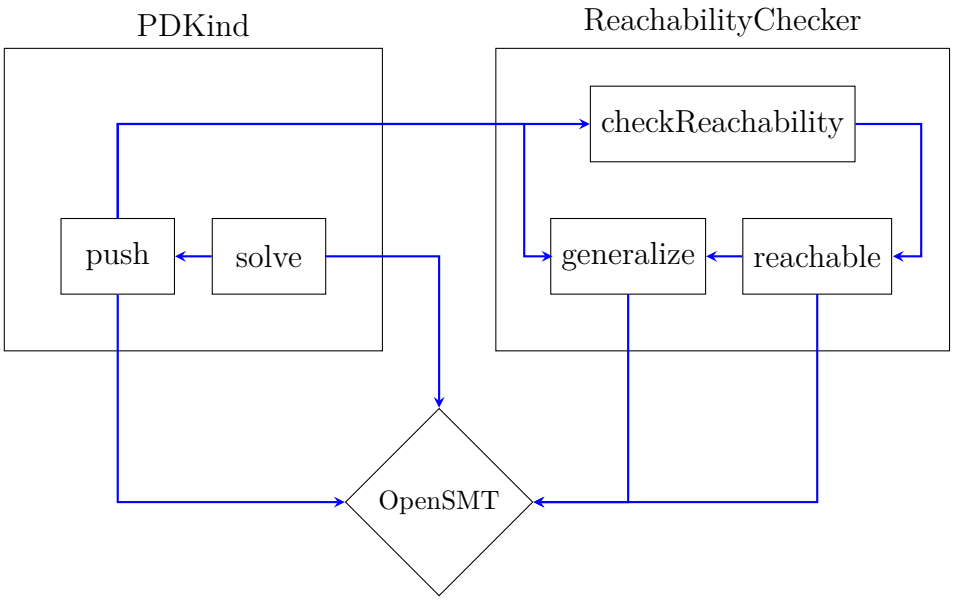

Figure 5.3 Architecture of PDKind engine

5.2.1 ReachabilityChecker class

The reachability procedure, shown in algorithm 4.2, determines whether a given state is reachable within a specific number of steps from the initial states. This check is performed using a depth-first search (DFS) strategy, along with several supporting methods and data structures that are crucial for the correctness and efficiency of our implementation.

In particular, we analyze the following components: **Satisfiability checking**, **Generalization**, **Explanation** and **Reachability frames** representation.

Satisfiability Checking

For satisfiability checking, we could choose from various SMT solvers, including Z3[16], cvc5[1], or OpenSMT[6], each offering different tradeoffs in terms of theory support and performance.

Since Golem already uses OpenSMT in its existing engines and provides a wrapper around it, we chose to use it in our implementation as well. This ensures consistency across the project and avoids the need to integrate a new solver. More importantly, using the same solver allows for a fair performance comparison between PDKind and other Golem engines, as any observed differences are then due to algorithmic behavior rather than backend solver performance.

OpenSMT also provides two essential features for our implementation: model generation and the ability to compute interpolants. The latter is particularly important for strengthening inductive invariants. Leveraging these built-in features allows us to avoid implementing additional mechanisms, making OpenSMT the most convenient and efficient choice for our engine.

Generalization

Generalization plays an essential role in PDKind by preventing redundant checks and improving abstraction. It allows the algorithm to reason about sets of

states rather than individual states. However, Golem does not provide a built-in generalization method.

Fortunately, it offers the components needed to construct our own method, as shown in algorithm 5.4. The generalization is performed by removing non-state variables from a formula using the `KeepOnly` function. Given a formula of the form $T \wedge C$, where T is a transition formula and C is a state formula representing a counterexample, we extract only the parts over state variables $\vec{x}$. This yields a generalized formula G, which satisfies the generalization conditions in definition 10.

1: **Input:** Model M, transition formula T, state formula C
2: **Output:** Generalized formula G
3: $StateVars \leftarrow$ GetStateVars()
4: $G \leftarrow$ KeepOnly(StateVars, $T \wedge C$, M)
5: **return** G

Figure 5.4 Generalize method

Notation. In algorithm 5.4, we work with the following inputs:

- T **(Transition formula)**: Defines the system's state evolution.
- C **(Counterexample formula)**: Represents a bad state discovered by the algorithm.
- M **(Model)**: A concrete assignment that satisfies $T \wedge C$ and shows how the system could reach the bad state.

From the model M, we construct a formula G that over-approximates it while being expressed only over state variables. This serves as a generalized lemma that can be further refined or blocked.
The implementation of this method uses Golem's `ModelBasedProjection` utility to perform the variable elimination and projection.

Although the `generalize()` method is used by both the `reachable()` function and the `push()` procedure, we placed it inside the `ReachabilityChecker` class. Since this class is already responsible for generating interpolants (i.e., explanations), it was a natural place to handle generalizations as well.

KeepOnly method

The `KeepOnly` method extracts only the state variables from the formula $T \wedge C$ by removing all auxiliary variables, specifically the intermediate states $\vec{w}$ and successor states $\vec{y}$, as described in definition 10. This ensures that the resulting generalized formula $G(\vec{x})$ depends only on the current state variables. As a result, G becomes an over-approximation: it captures not just the specific assignment given by model M, but all current states that satisfy $T \wedge C$.

Satisfying the Generalization Definition

To confirm that algorithm 5.4 satisfies the generalization definition, we verify the two required properties:

1. **Over-approximation:**
    - By removing $\vec{w}$ and $\vec{y}$, we ensure that $G(\vec{x})$ abstracts all possible satisfying assignments of the transition relation T and condition C.
    - This directly satisfies the first requirement in Definition 10.
2. **Consistency with Initial States $A(\vec{x})$:**
    - Although A is not directly present in the generalization formula, definition 10 requires that the generalization remains consistent with it.
    - Since we only remove variables and do not add constraints, $G(\vec{x})$ does not exclude any valid states reachable from the initial condition $A(\vec{x})$.
    - This guarantees that $G(\vec{x})$ does not eliminate any valid initial-state successors and satisfies the second requirement in the definition.

Thus, algorithm 5.4 produces a valid and useful generalization method for PDKind.

Explanation

Instead of implementing a custom $Explain()$ method, we utilize the OpenSMT solver feature, which is an interpolation. To obtain the interpolant, we tell the solver to compute an interpolation of the first two formulas inserted in it. For example, given a satisfiability check $CheckSat(A \wedge B \wedge C)$, we want an interpolant of $A \wedge B$.

Definition 12. **Craig Interpolation [8]:** Let A and A' be first-order formulas such that $A \models A'$. If A and A' share at least one predicate symbol, then there exists a formula I (called an interpolant) such that:

- $A \models I$
- $I \models A'$
- All predicate symbols in I occur in both A and A'

The use of interpolation as an explanation step is directly supported by the design of PDKind. As described in [12], interpolants are used to block infeasible counterexamples in a way that preserves soundness. The interpolant overapproximates the set of states from which a violation could occur but is still inconsistent with the property being violated. In our implementation, when a spurious counterexample is detected, the interpolant we extract serves as a weakening of the generalization that excludes the infeasible trace. Because the interpolant is implied by the reachable prefix and inconsistent with the bad state, it blocks the counterexample while preserving states that are still potentially valid. This makes it a suitable and valid explanation in the PDKind algorithm, consistent with the original paper.

On line 13 in Algorithm 4.2, we require an interpolant from the $CheckSAT()$ call that occurred in the previous *Reachable*() call. To support this behavior, we modify our *Reachable*() method on line 17 to return an interpolant along with the $false$ result when a check fails. Since interpolation is only defined when the formula is unsatisfiable, we only request an interpolant if $CheckSAT()$ returns $false$.

In our implementation, each $CheckSAT()$ is performed using a separate solver instance. This means obtaining an interpolant is simple; we instruct the solver instance to return the interpolant generated from its internal state, reflecting the formulas used in the failed check.

Reachability frames

For the $Reachabilityframes$ representation, we had several choices. The simplest approach was to create a list R of formulas, where each frame is stored as $R_i := R[i]$. Each time we call *Reachable*() from outside, we would construct a new list of reachability frames.

While this method is straightforward, it is inefficient because it discards previously computed frames with each call, preventing reuse and requiring redundant computation.

To address this, we implement a *Reachability* class that maintains a persistent list of frames. Each call to *Reachable*() extends the existing list rather than creating a new one. This way, we can reuse previously computed information across multiple calls and avoid redundant recomputation.

As mentioned in the paper [12], we need a structure that holds formulas representing the set of k-reachable states from the initial states, for all $k \in \{1, \ldots, n\}$. We also need to access and update these formulas by index. For this purpose, we define the structure `RFrames`, which maintains a vector `r` of formulas `PTRef`, where $r[i]$ is the over-approximation of states that can reach the target in exactly $k - i$ steps, or equivalently, states from which the target is reachable in $k - i$ transitions.

Inserting a formula `f` is creating a conjunction of `f` and `r[i]` and storing it in `r[i]`. As shown in Figure 5.5, 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 5.5 RFrames structure

Implementation

Now, we can move on to the actual implementation of the `ReachabilityChecker` class, which brings together the components discussed in the previous sections.

The `ReachabilityChecker` class encapsulates the mechanism needed for performing reachability checks. As analyzed earlier, creating a new instance of the `RFrames` structure for each reachability check would be inefficient. Therefore, we maintain a single instance of `RFrames` as a member of the `ReachabilityChecker` class and reuse it across all reachability queries.

Although the `ReachabilityChecker` class could be placed in its own file, we follow the structure of Golem and include it in the engine file alongside the rest of the code.

Reachable function:

The `reachable(unsigned k, PTRef formula)` function is the core of the `ReachabilityChecker` class. It implements the pseudocode shown in algorithm 4.2, but for the function to integrate correctly with the rest of the engine, several modifications are necessary.

First, we must return a formula alongside the reachability result. If the `reachable()` function returns `false`, we ask the solver to produce an interpolant and return it together with the result. Otherwise, we return `True` and a `false` term.

To generate interpolants, we need to set up a solver config at the beginning of

the function. This config is then used to initialize solvers for both satisfiability checking and interpolant generation.

The first solver is used when `k = 0`, to check whether the given formula holds in the initial states. If this check returns `false`, we proceed to generate the interpolant as shown in figure 5.2, using `mask = 1`. This is because we do not insert the transition formula into the solver, so we effectively compute an interpolant for $A \wedge B$ with respect to A.

If the given `k` is greater than zero, we proceed with the while loop. Before doing so, we must shift the version of the input `formula` from 0 to 1. We do this using the `TimeMachine` and its `sendFlaThroughTime(PTRef fla, int steps)` function, which increments the formula version by the given number of steps.

Versioning is crucial here because we are verifying whether the `formula` can be reached via a single transition from a specific RFrame. Therefore, formulas added to the solver must follow the structure:

$$(R[k-1]_0 \wedge T_{0,1} \wedge formula_1)$$

As we will observe, each `R[i]` is consistently versioned to 0.

In section 5.2.1, we analyzed that the `Explain()` method on line 13 in algorithm 4.2 is omitted, and instead, the interpolant is returned by the `Reachable()` function, which is called on line 10.

The pseudocode currently concludes after these steps, but an additional task remains: generating the interpolant when `k > 0`. In the while loop, if the satisfiability check fails, we request an interpolant from the solver in the same way as shown in figure 5.2. However, the interpolant must be shifted back by one version using the `TimeMachine`, ensuring it has version 0. This aligns with our earlier note that all `R[i]` are versioned to 0, and this interpolant will be used to update them.

Yet, this interpolant alone is not sufficient. We must also verify whether the given `formula` holds in the initial states. If it does, we return the interpolant. If it does not, we create another interpolant for the initial state check as we did in the `k = 0` case, and return the disjunction of the two interpolants.

CheckReachability function: In the Push procedure, we need to check whether a formula is reachable over a range of steps, rather than a fixed number. To support this, we define an extended version of the `reachable()` function, which takes a step range `(k_from, k_to)` and iterates through the values using a loop. For each k, we call `reachable(k, formula)` and return the result as soon as we encounter a successful check. If none of the calls succeed, the function returns the result of the last failed call, including the number of steps used and the interpolant generated by the last solver instance. This interpolant serves as an explanation of why the formula is unreachable.

We initialize a new solver instance for each reachability check. Although this introduces some performance overhead, it is necessary to keep model and interpolant generation isolated, avoiding side effects from previous solver states. This approach ensures correctness, especially when extracting models and interpolants using OpenSMT, which does not return models by default unless configured. It also allows us to efficiently find the earliest reachable step, improving precision in the overall algorithm.

5.2.2 Push procedure

The `push()` procedure, which implements algorithm 4.3, is the core of our solution. It connects all parts of our engine to either produce inductive strengthening or find a counterexample.

Induction Frame

The induction frame is a key data structure in the PDKind algorithm. It is used to store lemmas that lead to constructing inductive invariants. The induction frame is also discussed in the paper [12] and formally defined in section 11.

In our implementation, we represent the *Induction Frame* as a set of objects, where each object consists of:

- A lemma, representing an inductive candidate formula.
- A counterexample structure, which stores additional information about the counterexample.

Each `IFrameElement` holds a `PTRef lemma` and a `CounterExample counter_example`. We use the dedicated `CounterExample` structure instead of a plain `PTRef` because it allows us to store both the formula and the number of steps required to reach it from the initial states. This additional data is crucial for generating unsatisfiability witnesses when proving that a property is UNSAFE.

In our solution, we implement the frame as a set of `IFrameElement` instances stored in the `InductionFrame` structure.

Transition Construction

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 take a conjunction of these two results 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 initialization

Return Values and Result Tracking

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

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

```
struct PushResult {
    InductionFrame i_frame;
    InductionFrame new_i_frame;
    int n;
    bool is_invalid;
    int steps_to_ctx;
    PushResult(InductionFrame i_frame,
               InductionFrame new_i_frame,
               int n,
               bool is_invalid,
               int steps_to_ctx) { ... }
};
```

Figure 5.7 PushResult structure

Witness Construction

In the next step, we must update the `steps_to_ctx` value, which was set to 0 during initialization. This update occurs in the same part of the code where we set the `isInvalid` flag to `True`. The new value 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 the value `k` returned by the `CheckReachability()` function. This tells us that `g_cex` is k-reachable from the initial states, so the final value of `steps_to_ctx` will be:

```
g_cex.num_of_steps + k
```

Finally, we need to assign the `num_of_steps` value for each newly created counterexample. We already analyzed in section 5.2.4 that a new counterexample is only generated once. This process is shown in figure 5.8.

The new counterexample is created by generalizing the previous potential counterexample and increasing its version by `k`. Therefore, the value of `num_of_steps` will be the same as the previous counterexample's `num_of_steps`, incremented by `k`.

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

Figure 5.8 CounterExample initialization

Model Generation

At last, we make a minor adjustment. In the pseudocode, the model is obtained directly from the `CheckSAT()` command. However, in our implementation, we request the model only after verifying that the result of `CheckSAT()` was positive. Therefore, the call to extract the model is placed after an `if` statement checking the satisfiability result.

5.2.3 PDKind Engine

In this section, we describe how we implemented the PDKind engine and integrated it into the Golem solver. The engine ties together all parts of the algorithm and runs the main solving loop.

The `PDKind` class inherits from the `Engine` class in Golem, which defines a virtual method `solve(ChcDirectedHyperGraph const & graph)`. In our implementation, we override this method to solve the transition system using the PDKind algorithm.

This method accepts a hypergraph that represents the verification task. First, we convert the hypergraph into a normal graph. To solve the normal graph, we overload the method `solve(ChcDirectedGraph const & system)`, which operates on this converted structure.

Within the overloaded method, we check whether the graph is trivial. If so, we use Golem's `Common` class and its `solveTrivial()` function to handle it. If the graph is non-trivial, we transform it into a `TransitionSystem` and pass it to `solveTransitionSystem(TransitionSystem const & system)`, which runs the PDKind algorithm.

SolveTransitionSystem method

This function receives a `TransitionSystem` as input, giving us access to the initial states, transition relation, and the query, which represents the set of bad states. The goal is to construct either an inductive invariant that proves safety or a counterexample that demonstrates unsafety. The property we attempt to prove is the negation of the query.

This function implements the pseudocode shown in algorithm 4.1. Before entering the main loop, we perform two preliminary checks using the `MainSolver`:

1. **Are the initial states empty?** If the set of initial states is empty, the system is trivially safe and we return $SAFE$.

2. **Does the query already hold in the initial states?** If yes, the system is immediately unsafe and we return $UNSAFE$.

If neither of these cases applies, we proceed with the main loop of the PDKind algorithm, which repeatedly calls the `push()` procedure to search for a k-inductive strengthening of the property P.

In our implementation, the `push()` procedure also provides additional information needed for generating witnesses. For satisfiability witnesses, we use the final `InductionFrame` produced by `push()` and first convert it into a k-inductive invariant. We then use Golem's `kinductiveToInductive()` function to transform it into an inductive invariant.

For unsatisfiability witnesses, we use the `steps_to_ctx` value returned by `push()` and store it in the `TransitionSystemVerificationResult`, which is then returned by the engine.

The method terminates when it either constructs a valid k-inductive invariant (and returns $SAFE$) or discovers a counterexample trace (and returns $UNSAFE$).

5.2.4 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, while 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 clauses.

- An **unsatisfiability witness** is a proof presented as a sequence of derivations of ground instances of the predicates.

Witnesses are essential in formal verification, as they provide concrete evidence to support the solver's conclusion, ensuring that verification results are both explainable and reproducible.

In Golem [5], each engine provides a validity witness when the option `--print-witness` is used. To follow this structure, we implemented such functionality in the PDKind engine as well.

UNSAT Witness

First, we describe the implementation of the unsatisfiability witness in our engine. The goal is to generate counterexample paths during the CHC solving process. These paths demonstrate that the safety property does not hold.

In the PDKind procedure, when a counterexample is detected, we track how many steps are required to reach it. Rather than storing entire traces, we only store the number of steps to the counterexample from the initial states. This keeps the representation compact while still allowing us to reconstruct the path when needed.

We encapsulate this data in a structure called `CounterExample`, which includes:

- the formula representing the counterexample
- the number of steps to reach it from the initial states

This information is attached to each tuple in the $Induction\ Frame$ as ($lemma$, $counterExample$). The lemma is valid up to a certain depth, and the counterexample it fails to refute becomes the next candidate in the push process.

In algorithm 4.3, a new counterexample g_2 is generated at line 18 and reused in the else branch starting at line 23. The number of steps assigned to g_2 is computed by taking the number of steps associated with the previous counterexample and adding k, which is the depth of the current reachability check.

To make this work in the implementation, the `Push()` procedure was extended to return an additional field: the total number of steps to the final counterexample. On line 21, when a reachable counterexample is found, this number is returned as part of the witness structure.

Finally, the `solveTransitionSystem()` function records this value in the `TransitionSystemVerificationResult`, allowing Golem to generate a full counterexample trace using built-in utilities when the witness is requested.

SAT Witness

Now we describe the construction of a satisfiability witness in the PDKind engine. Here, the goal is to produce a valid inductive invariant that proves the safety property.

As described in algorithm 4.3, the $Push()$ procedure constructs an `Induction Frame`, a set of tuples $(lemma, counterExample)$. The lemmas in this frame are valid up to n steps and jointly refute all known counterexamples. When the frame stabilizes (i.e., $F = G$ at line 11), no new lemmas are needed, and we have a k-inductive invariant.

At this point, the PDKind algorithm concludes by returning a `SAFE` result (line 12 in algorithm 4.1).

To construct the witness, we form a conjunction of all lemmas in the final $Induction\ Frame$. This yields an n-inductive invariant that proves the property for n steps. However, to obtain a classical inductive invariant (valid at all depths), we pass this result to Golem's helper function `kinductiveToInductive()`, which transforms the n-inductive invariant into a standard inductive invariant.

This invariant is then returned as part of the `TransitionSystemVerificationResult` and printed when the user requests the witness.

Summary

Validity witnesses not only validate the result of the solver but also increase the trustworthiness and usefulness of the tool. In PDKind:

- UNSAT witnesses are derived from tracked steps to a counterexample and reconstructed using Golem's trace utilities.
- SAT witnesses are built from the final stable induction frame and converted to inductive invariants.

This integration ensures the PDKind engine aligns with the broader structure and expectations of the Golem solver ecosystem.

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 an algorithm based in IC3/PDR which PDKind modifies by replacing induction with k-induction, and KIND is the direct application of k-induction on safety property for transition system.

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.

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

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 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.

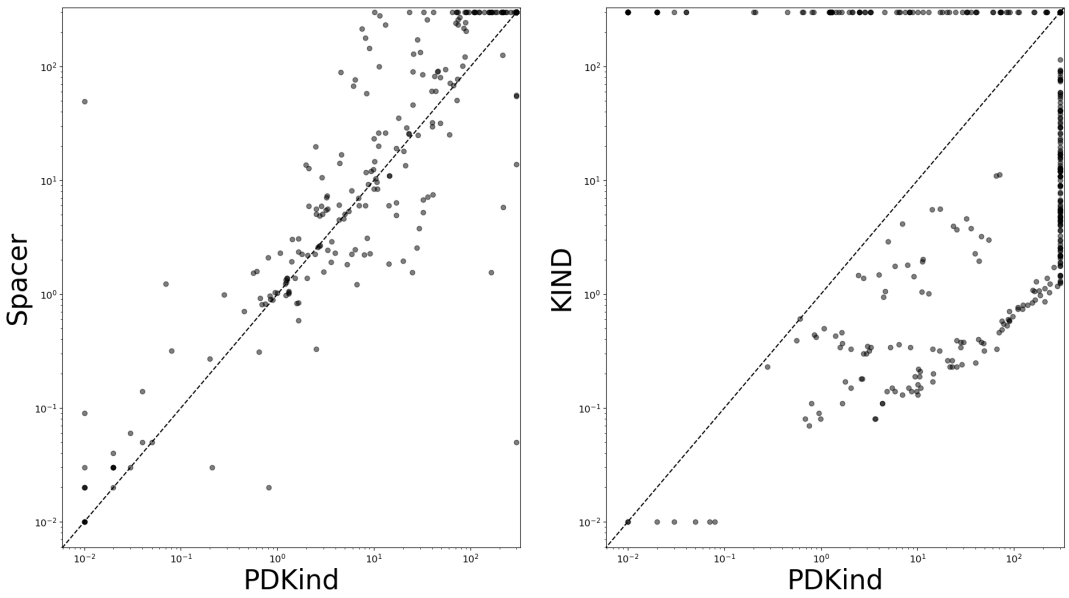

Figure 6.2 Time Comparison: SAT results

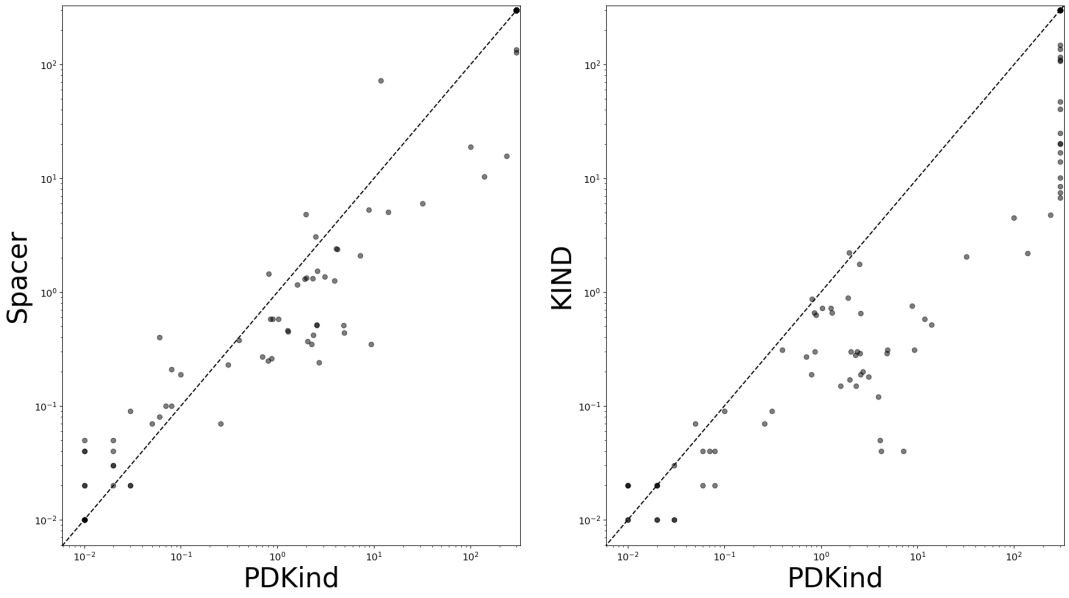

Figure 6.3 Time Comparison: UNSAT results

6.3 Discussion

The results show that PDKind performs well overall, but still falls behind KIND in the total number of solved benchmarks. KIND solves the most, followed by PDKind, then Spacer.

PDKind is especially strong on SAT problems. It solves more satisfiable benchmarks than Spacer and comes close to KIND. It even solves 11 SAT benchmarks that neither of the other two engines manage to solve. This highlights the strength of its k-inductive reasoning, generalization, and interpolation in proving satisfiability, particularly in cases where the others time out.

On UNSAT problems, PDKind is noticeably weaker. It solves fewer cases than both KIND and Spacer, often timing out where the others succeed. This suggests that while PDKind's separation of reachability and induction gives it flexibility, it doesn't always match the robustness of IC3-style in Spacer or the simpler approach in KIND.

In fact, PDKind solves 97 benchmarks that KIND doesn't, further demonstrating its complementary value. While KIND has the highest overall success rate, these results show that PDKind is not just a weaker variant but brings in unique strength on many benchmarks. This makes a strong case for including PDKind as a core part of Golem's multi-engine strategy.

Overall, PDKind is reliable on satisfiable problems and shows promise as a complete solver. While it still needs improvement on harder UNSAT cases, its ability to solve problems that other engines miss makes it a useful addition to the Golem lineup.

7 Conclusion

In this work, we introduced 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, especially on SAT benchmarks. While PDKind did not reach the top performance in total solved instances, it successfully solved several problems that no other engine could, showing its value as a complementary solver in Golem's engine portfolio.

We believe that this engine presents a well-performing alternative to other engines. Its strengths on certain benchmarks and its unique problem coverage make it a useful addition, and its usefulness is expected to grow with further improvements.

Bibliography

1. Barbosa, Haniel; Barrett, Clark; Brain, Martin; Kremer, Gereon; Lachnitt, Hanna; Mann, Makai; Mohamed, Abdalrhman; Mohamed, Mudathir; Niemetz, Aina; Nötzli, Andres; Ozdemir, Alex; Preiner, Mathias; Reynolds, Andrew; Sheng, Ying; Tinelli, Cesare; Zohar, Yoni. cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman, Dana; Rosu, Grigore (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2022, pp. 415–442. isbn 978-3-030-99524-9.

2. Barrett, Clark; Fontaine, Pascal; Tinelli, Cesare. *The Satisfiability Modulo Theories Library (SMT-LIB)* [`www.SMT-LIB.org`]. 2016.

3. 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.

4. Bjørner, Nikolaj; Gurfinkel, Arie; McMillan, Ken; Rybalchenko, Andrey. Horn Clause Solvers for Program Verification. In: *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Beklemishev, Lev D.; Blass, Andreas; Dershowitz, Nachum; Finkbeiner, Bernd; Schulte, Wolfram. Cham: Springer International Publishing, 2015, pp. 24–51. isbn 978-3-319-23534-9. Available from doi: `10.1007/978-3-319-23534-9_2`.

5. 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`.

6. 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.

7. 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`.

8. Craig, William. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic* [online]. 1957, vol. 22, no. 3, pp. 269–285 [visited on 2025-04-20]. issn 00224812. Available from: `http://www.jstor.org/stable/2963594`.

9. 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.

10. Holzmann, G.J. The model checker SPIN. *IEEE Transactions on Software Engineering*. 1997, vol. 23, no. 5, pp. 279–295. Available from doi: 10.1109/32.588521.

11. 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.

12. 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.

13. 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.

14. 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.

15. 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.

16. Moura, Leonardo de; Bjørner, Nikolaj. Z3: An Efficient SMT Solver. In: Ramakrishnan, C. R.; Rehof, Jakob (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. isbn 978-3-540-78800-3.

17. 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.

18. 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