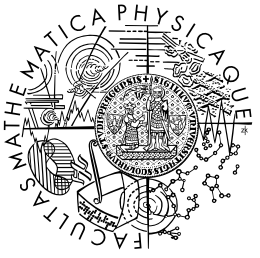

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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: 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 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 (CHCs)[9] 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[6] is one such solver, which integrates the interpolating SMT solver OpenSMT[10]. 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, whether 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)[11]. PDKind is an IC3[17]-based algorithm, that separates reachability checking and induction reasoning, allowing the induction core to be replaced 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[6].

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

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 [11]:** 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 [11]:** State s has a successor s', when $T(s(\vec{x}), s'(\vec{x}'))$ is true.

Definition 3. **k-Reachability [11]:** 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 property 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. This property is crucial in verifying the correctness of programs and systems modeled using transition systems.

Definition 4. **Safety Property:** A property P is a **safety property** if it ensures that "nothing bad will ever happen" during the execution of the system. Formally, given a transition system $(S, \mathcal{F}, \mathcal{T}, \mathcal{I})$, a safety property P holds if for every reachable state $s \in S$, the condition $P(s)$ is satisfied.

Definition 5. **Invariant [11]:** An **invariant** is a property that holds in all reachable states of a transition system. Formally, a property I is an invariant if, for a transition system $(S, \mathcal{F}, \mathcal{T}, \mathcal{I})$, every reachable state $s \in S$ satisfies $I(s)$.

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

Verifying Safety: To verify whether a transition system satisfies a given safety property, we need to check whether all reachable states of the system satisfy the property. This can be done using different approaches:

- **Explicit State Exploration:** Enumerate all reachable states and verify that each state satisfies P.
- **Bounded Model Checking (BMC):** Unroll the transition system up to a given bound k and check whether an unsafe state is reachable within k steps.
- **Inductive Invariant Checking:** Find an inductive invariant Inv such that $\mathcal{I} \Rightarrow Inv$, $Inv \wedge \mathcal{T} \Rightarrow Inv'$, and $Inv \Rightarrow P$. If such an invariant exists, then P holds for all reachable states.

Example: Consider a transition system modeling a counter with the following transitions:

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

A safety property might state that the counter never exceeds a threshold, e.g., $x \leq 10$. The verification process would involve checking whether there exists a trace that leads to a state where $x > 10$. Which, in this situation, obviously exists.

Ensuring safety is critical for system reliability, and we can use various formal verification techniques to determine whether a system satisfies its safety properties.

2.3 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 [8]. 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.

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) 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:

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

Safety clause:

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

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. To obtain a finite trace of the loop, we unroll it k times, where k is a finite number.

$$Init(V) \wedge Tr(V, V^2) \cdots \wedge Tr(V^{k-1}, V^k) \wedge Bad(V^k) \tag{2.5}$$

This unrolled loop is known as a **Bounded Model Checking (BMC)** formula. By increasing k, we can evaluate more BMC formulas, which helps detect more system bugs. However, we must identify inductive invariants to prove that no bugs exist. In our work, we will leverage BMC formulas to discover these invariants.

2.5 OpenSMT

OpenSMT[7] 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[10], 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[17] is a commonly used method that uses induction to show a property is invariant by incrementally constructing an inductive strengthening of the property. PDKind [11] 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[11]): Let $\mathcal{F}$ be a set of state formulas, where $P \in \mathcal{F}$. Then P is $\mathcal{F}$-inductive if:

$$\begin{aligned} I(\vec{x}) &\implies \mathcal{F}(\vec{x}), && \text{(init)} \\ \mathcal{F}(\vec{x}) \wedge T(\vec{x}, \vec{x}') &\implies P(\vec{x}'). && \text{(cons)} \end{aligned}$$

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

[11] 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[11]):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 $Definition$ 7, we can see that a property that is inductive is 1-inductive and also k-inductive for any k. In the other direction, [4] 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, [5] shows that k-induction can yield more succinct strengthening. For other theories, such as *Booleanlogic* or *Lineararithmetic*, k-induction may be exponentially more powerful than induction due to weaker interpolation.

Practical Effectiveness

[11] shows that k-Induction is effective, especially when combined with algorithms like IC3. [11] 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 [6] 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[10], 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

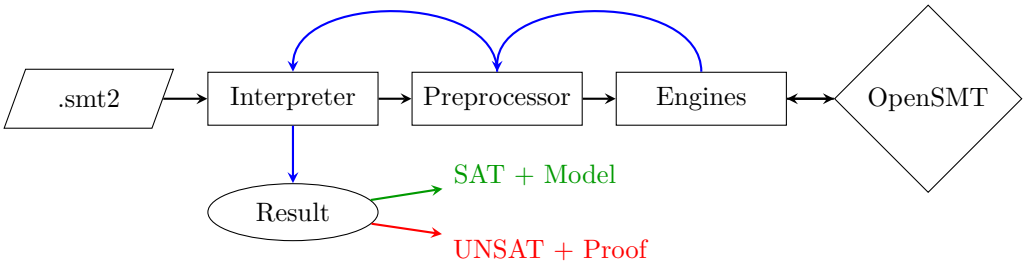

Figure 3.1 Architecture of Golem

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) [16]
- Interpolation-based Model Checking (IMC) [13]
- Lazy Abstractions with Interpolants (LAWI) [14]
- Spacer [12]
- Transition Power Abstraction (TPA) [6]

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[17]-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 [11]. After reading this chapter, the reader should be able to understand the parts of the PDKind algorithm and orientate in our implementation.

4.1 Reachability checking procedure

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

Let us have a formula in the form

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

Definition 8. 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 9. **(Interpolant)** [11] 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 10. **(Generalization)** [11] 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.

Method shown in Algorithm 4.1 tries to reach the initial states backwards by using a depth-first search strategy.

To check if F is reachable from the initial states in k steps, we first check whether F is reachable in one transition from the previous frame R_{k-1}. If there is no such transition, then F is not reachable in k steps. Otherwise, we get a state that satisfies R_{k-1} and from which F is reachable in a single step.

We then call a generalization procedure, which produces a formula G (see Definition 10). Here, generalization helps avoid redundant checks by creating an

```
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 4.1 Reachable method

abstraction of the counterexamples. Instead of storing the given state, it generates a formula covering multiple possible counterexamples, which reduces the number of times the reachability procedure gets called.

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 terminates. Otherwise, we can learn an explanation through interpolation (see Definition 9) and eliminate G by adding the explanation into the frame R_{k-1}.

4.2 Push procedure

Definition 11. **(Induction Frame)** [11] 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.

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 procedure shown in Algorithm 4.2 is the core of the PDKind algorithm. It takes the invariant and tries to construct a stronger inductive proof by analyzing its counterexamples and strengthening constraints, while switching between

generalization and reachability checking. We will break it down into smaller pieces to understand how it works.

The first step (line 11) is to check whether *lemma* is k-inductive (see Definition 7). This is done by checking if the formula $(F_{ABS} \wedge T_k \wedge \neg lemma)$ is satisfiable. We can notice, that $CheckSAT$ call returns model m_1 which is an assignment of values to variables that satisfies the formula. 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. Otherwise, we get the model m_1 and save it for later.

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

On line 19, we can see that $Reachable()$ accepts more arguments and returns more values than shown in Algorithm 4.1. This new $Reachable(i, j, F)$ method checks if F is reachable in k steps where $i \leq k \leq j$. If this call wasn't successful, we get an interpolant i_1 and assign it to g_3 which eliminates g_2. We found a new induction obligation (g_3, g_2), which is a strengthening of F. Now, we can try again with a potential counterexample eliminated.

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

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 4.2 Push procedure

4.3 PD-Kind procedure

The main PDKind procedure shown in Algorithm 4.3 checks if property P is invariant by iteratively calling the $Push$ procedure to find a k-inductive strengthening of P for some $1 \leq k \leq n+1$. PDKind iteratively strengthens the invariant until it becomes inductive. The strengthening G is k-inductive, and if $F == G$, then no more strengthening is required, and P is proven to be invariant ,so we return $SAFE$. If the $Push$ procedure marks $isInvalid$ as $True$, then a counterexample has been found, proving that the property is not invariant and we return $UNSAFE$. Otherwise, we update n and repeat the loop.

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: $n \leftarrow 0$
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 4.3 Main PD-Kind procedure

5 Implementation

In this chapter, we will refer to the description of the PDKind algorithm, analyze it and modify it to fit our implementation.

5.1 Reachability checking procedure

The reachability procedure, shown in 4.1 is a procedure, which determines whether a given state is reachable in a specific number of steps from a set of initial states. This check is completed using a depth-first search (DFS) strategy, along with other methods and data structures that need to be analyzed to ensure correctness and efficiency in our implementation.

The choice of depth-first search (DFS) in this procedure balances efficiency and memory usage. While DFS minimizes memory overhead compared to breadth-first search (BFS), it may explore deeper infeasible paths first, potentially increasing runtime in cases where a shallower counterexample exists.

The above-mentioned methods and data structures, that need analysis are: **Satisfiability checking**, **Generalization**, **Explanation** and **Reachability frames representation**.

Satisfiability Checking

We could use various SMT solvers for satisfiability checking, including Z3[15], CVC5[1], or OpenSMT[7], with each solver having different strengths in terms of efficiency, theory support, or integration capabilities.

Since Golem already provides a wrapper around the OpenSMT solver and uses it for satisfiability checking in every other engine, we also chose it for our implementation. This choice maintains consistency across the whole project and avoids problems of integrating a new solver into the project.

Another reason for using OpenSMT is that since it is used in all other Golem engines, it provides a fair performance comparison of these engines with our PDKind engine. Using a different SMT solver in our implementation could give our engine an advantage or a disadvantage in terms of efficiency. By using OpenSMT, we ensure that these possibilities are isolated and that the observed performance differences would come mainly from algorithmic strength rather than solver efficiency.

Furthermore, OpenSMT provides utilities that are very important for our engine. One of them is providing models for satisfiability checks, and the other is the ability to generate interpolants. Since interpolation is used to strengthen inductive invariants, utilizing these features eliminates the need for additional mechanism implementation. While other SMT solvers may offer similar capabilities, choosing OpenSMT is the most convenient and efficient choice for our implementation.

Generalization

Generalization plays an essential role in PDKind by preventing redundant checks and improving abstraction. It ensures that the algorithm abstracts wider state representation rather than handling individual states separately. However, Golem does not provide a built-in generalization method.

Still, it has needed components to create our method, as shown in pseudocode Algorithm 5.1. The method uses the KeepOnly function to eliminate all non-state variables from the formula $T[B]^k(\vec{x}, \vec{w}, \vec{y}) \wedge C(\vec{y})$, retaining only the state variables $\vec{x}$. We will show that this approach satisfies the generalization definition 10.

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 5.1 Generalize method

Relationship between M, T and F

In algorithm 5.1, we use three formulas:

- T **(Transition formula)**: Defines the evolution of states over time.
- F **(State formula)**: Represents the state that we are analyzing.
- M **(Model)**: A satisfying assignment to formula $T \wedge F$ that provides an example of a state transition.

Given a model M, our goal is to abstract it into a formula G that describes a wider set of possible states.

KeepOnly method

The KeepOnly method extracts only the state variables from $T \wedge G$ by removing all the auxiliary variables, such as the intermediate states $\vec{w}$ and the successor states $\vec{y}$ from definition 10. This approach ensures that the generalized formula G depends only on the current state $\vec{x}$, which makes the formula $G(\vec{x})$ an over-approximation as it represents all possible states that satisfy $T \wedge F$ rather than just the single assignment given by model M.

Satisfying the Generalization Definition

To confirm that Algorithm 5.1 correctly implements generalization, we check the two required properties:

1. **Over-approximation**:
    - By removing $\vec{w}, \vec{y}$, we ensure that $G(\vec{x})$ captures all possible satisfying assignments for the transition relation T and condition F.
    - This directly satisfies the first condition of the generalization definition 10.
2. **Consistency with Initial States** $A(\vec{x})$:
    - Since we only eliminate intermediate variables and do not introduce new constraints, $G(\vec{x})$ remains consistent with the original set of reachable states.
    - This ensures that $G(\vec{x})$ does not eliminate any valid states from consideration.

Thus, algorithm 5.1 produces a valid and useful generalization 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 $A \wedge B$.

Interpolation is useful here because it provides a formula that separates $A \wedge B$ from C, ensuring that the learned constraints eliminate infeasible states without being too restrictive.

On line 13 in Algorithm 4.1, 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 representation

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. Instead, we create a more efficient approach by making a $Reachability$ class.

Each instance of the *Reachability* class maintains a persistent list of frames, where calling *Reachable*() on that instance extends the existing reachability frames instead of discarding them. This allows us to reuse previously computed information across multiple calls, improving efficiency and reducing unnecessary recomputation.

5.2 Push

Induction frame

Induction frame is a key data structure in the PDKind algorithm. It is used to store lemmas lead to creating inductive invariants.

In our implementation, we represent the $InductionFrame$ 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.

We choose to store the counterexample in a different structure, which allows us to keep not only the counterexample formula but also the number of steps needed to reach it from the initial states. This additional data is crucial for generating unsatisfiability witnesses when proving an $UNSAFE$ property.

Extended Reachable method

We previously mentioned that OpenSMT[**OpenSMT**] provides the option to retrieve a model after a successful satisfiability check. Since OpenSMT does not return models by default, we initialize a new solver instance for each satisfiability check. This ensures that model extraction remains independent for each query and avoids unintended side effects when performing multiple checks.

On line 19, we see that *Reachable*() accepts more arguments and returns more values than shown in Algorithm 4.1. The extended method $Reachable(i, j, F)$ now checks whether F is reachable within k steps, where $i \leq k \leq j$.

To implement this behavior, we create a wrapper function that iterates over possible values of k, calling $Reachable(k, F)$ in a loop. The function returns the first k where reachability is successful, along with the result.

If none of the calls succeed, instead of returning just $false$ result, we extract an interpolant from the final *Reachable*() check. This allows us to refine our constraints even in cases where reachability fails, improving the overall efficiency of the algorithm.

While this approach ensures correctness, initializing a new solver instance for each check introduces some performance overhead. However, since each $CheckSAT()$ call operates independently, this trade-off is necessary to maintain solver consistency. Similarly, iterating over multiple values of k increases computational effort but allows us to efficiently find the earliest reachable step, which improves overall precision.

5.3 PDKind

In our implementation, the property P is defined as the negation of a query, that we get on the input, which represents a set of bad states. This transformation ensures that proving P invariant is equivalent to proving that the original query is never reachable.

Before running the main loop of PDKind, we perform two initial checks:

1. **Are the initial states empty?** If there are no initial states, the system is then trivially safe, and so we return $SAFE$.

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

If neither of these conditions is met, the algorithm continues with an iterative process which calls $Push()$ procedure to find a k-inductive strengthening of P .

The method continues until it either finds a k-inductive invariant (proving safety) or discovers a counterexample (proving unsafety).

5.4 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, 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.

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 [6], 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 generate counterexample paths, we utilize a function in the Golem solver, which computes the path to a counterexample based on the number of steps required to reach the counterexample.

Instead of storing complete counterexample traces, we only keep track of the number of steps needed to reach counterexample for each potential counterexample we encounter during the CHC solving. The number of steps 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 number of steps to reach the counterxample.

In the next step, we need to correctly assign the number of steps to a counterexample to each potential counterexample, that we find. In Algorithm 4.2, 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 number of steps to a counterexample assigned to g_2 is the number of steps to a counterexample in $counterExample$ incremented by k.

Finally, we need to modify $Push()$ procedure to return the number of steps to a counterexample. On line 21, when we encounter a reachable counterexample g_2, we assign the steps to a counterexample returned by the $Push()$ procedure to be the number of 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 4.2, we construct an $InductionFrame$, 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 4.3 because we are constructing the satisfiability witness. We can then take the final $InductionFrame$ 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 $InductionFrame$, 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.

6 Realization

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.

6.1 Data structures

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

6.1.1 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 6.1.

```
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 6.1 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 6.2, 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 6.2 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.

6.1.2 PDKind structures

Reachability frames

As mentioned in the paper[11], 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 6.3, 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 6.3 RFrames structure

Induction frame

The induction frame is another structure mentioned in the paper[11] and defined in the section 11. 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.

6.2 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 6.4.

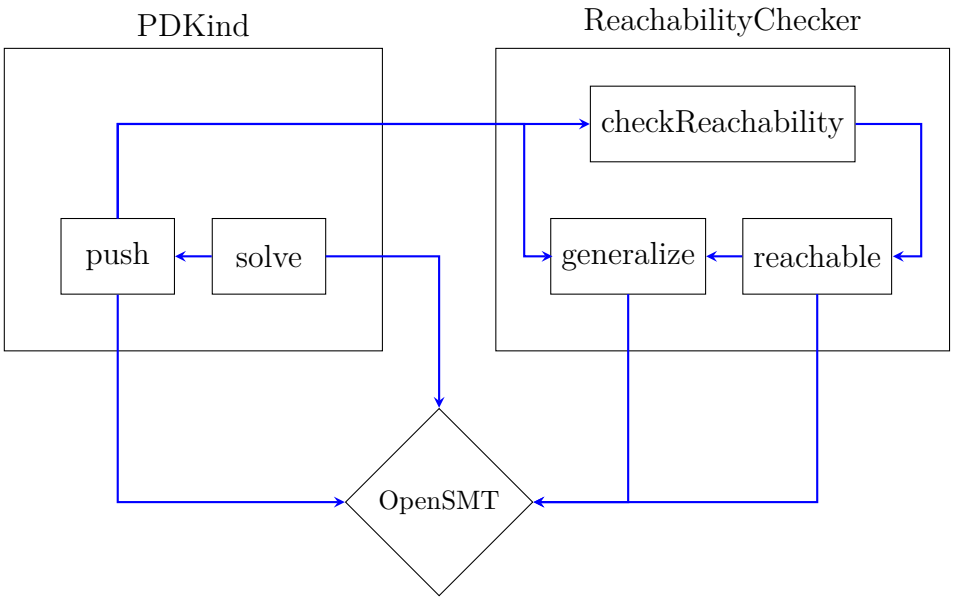

Figure 6.4 Architecture of PDKind engine

6.2.1 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 6.5 ReachabilityChecker class

The `ReachabilityChecker` class encapsulates the mechanism needed for reachability checking. As we analyzed in Section 5.1, 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 4.1. 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 6.2 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 5.1, we have already analyzed that the `Explain()` method on line 13 in Algorithm 4.1 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 6.2. 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 4.2, 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 5.1. 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.

6.2.2 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 4.3. 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 4.2 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 6.6, we show how we created the transition $T[F_{ABS}]^k$ which satisfies the definition shown in section 4.1 .

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 6.6 Transition initialization

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 6.7.

```
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 6.7 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 5.4 that a new counterexample is only generated once. This process is displayed in a code snippet in Figure 6.8.

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 6.8 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.

7 Experiments

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

7.1 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.

7.2 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 7.1, 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 7.1 Number of solved benchmarks from LRA-TS category

In Figure 7.2 (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 7.3, 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.

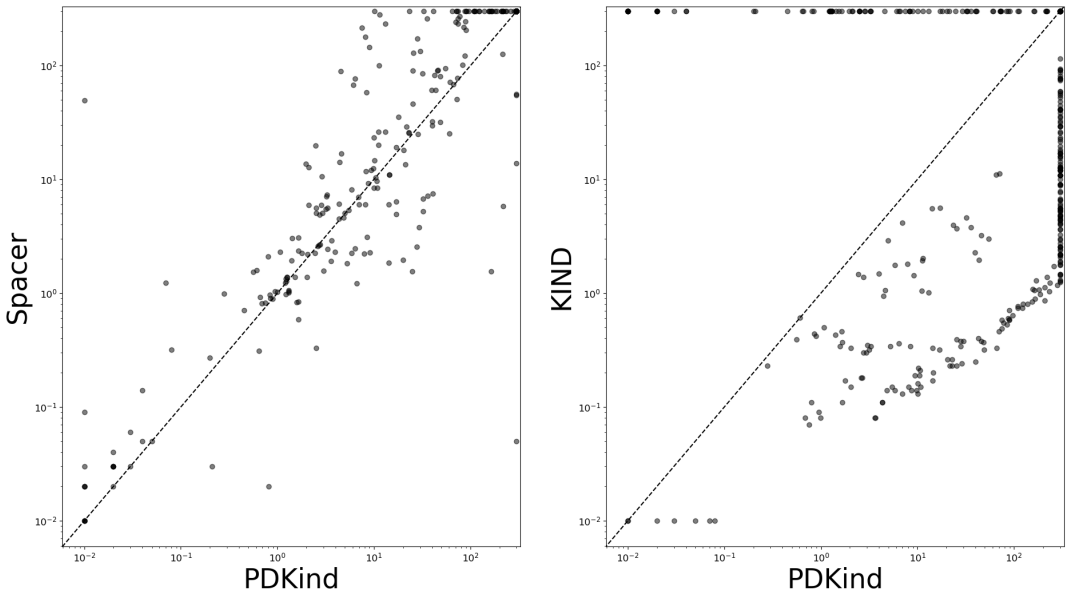

Figure 7.2 Time Comparison: SAT results

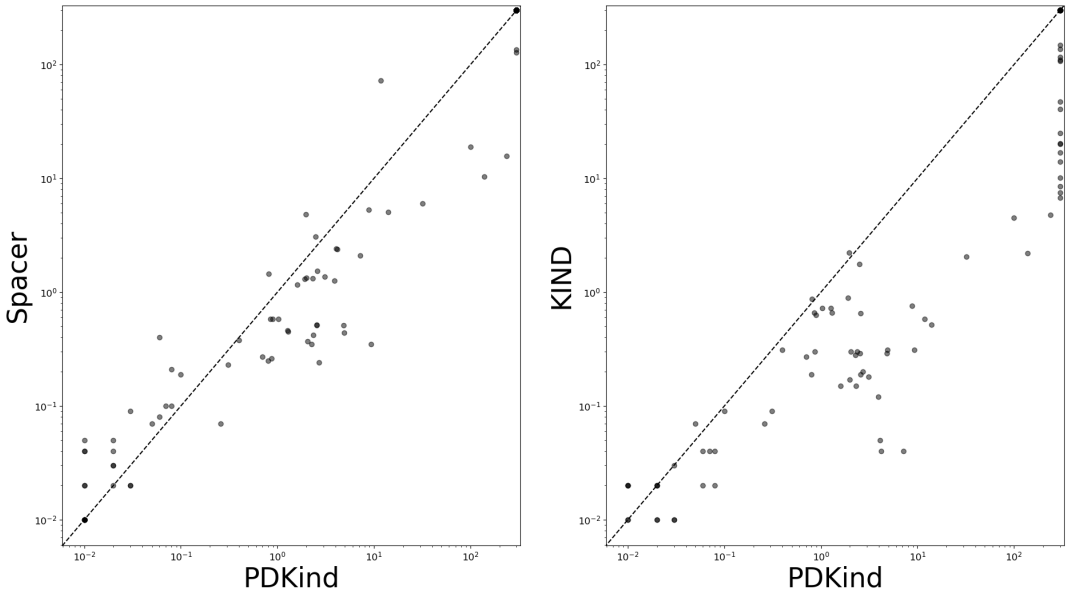

Figure 7.3 Time Comparison: UNSAT results

7.3 Discussion

The experiments show that PDKind outperforms Spacer in identifying SAT instances and matches it in identifying UNSAT instances, suggesting that its approach is particularly effective for proving existence results. However, it doesn't match the performance of KIND, which solves more problems overall.

The main weakness of PDKind appears in handling UNSAT cases, where it performs slightly worse than Spacer and significantly worse than KIND.

Another observation is that while PDKind is being generally slower than KIND, there are several cases where PDKind successfully solves problems that KIND times out on. This suggests that PDKind might handle certain problem structures more effectively, particularly in cases where KIND's approach does not scale well.

8 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. 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. Birgmeier, Johannes; Bradley, Aaron R.; Weissenbacher, Georg. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In: Biere, Armin; Bloem, Roderick (eds.). *Computer Aided Verification*. Cham: Springer International Publishing, 2014, pp. 831–848. isbn 978-3-319-08867-9.

5. 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`.

6. 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`.

7. 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.

8. 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`.

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. 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.

11. 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.

12. 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.

13. 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.

14. 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.

15. 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.

16. 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.

17. 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