Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Artificial Intelligence and Systems Engineering

Improving Model Checking Portfolio Efficiency through Partial Result Transfer

TDK REPORT

Author
Klevis Imeri

Advisor
Ádám Zsófia

November 13, 2025

Contents

Abstract

Software is an indispensable part of modern life, from smartphones to satellites, but such code inevitably contains bugs that require fixing. Software verification is the process of identifying bugs in computer programs or formally demonstrating their absence. A powerful verification technique is model checking, which, given a formal model of a program and a desired property, systematically traverses the entire state space to determine whether the property holds.

The primary challenge in model checking is the state space explosion, where the number of states can grow exponentially even in simple programs. To tackle the problem, a diverse range of algorithms are utilized, such as CEGAR (Counterexample Guided Abstraction Refinement), BMC (Bounded Model Checking), and k-Induction. Each technique possesses distinct strengths and weaknesses, excelling at different types of program complexities (e.g., loops, large data domains). Consequently, no single model checking algorithm is optimal for all situations.

To address the issue, strategies like algorithm portfolios have been proposed. A general approach to enhance algorithm portfolios is cooperation, which involves sharing information between different algorithms. A common cooperation strategy is to use sequential portfolios. When an algorithm is deemed insufficient for a given problem, it is stopped, and its partial results are extracted and shared with the next algorithm in the sequence. However, the methods currently used to pass partial results are often complex and present a significant implementation barrier. As a result, a simple way is proposed to extract invariants from CEGAR algorithms and pass them to other algorithms using the Control Flow Automaton (CFA).

Furthermore, while much work has focused on cooperation, the criteria for stopping an algorithm are often just simple static time or memory limits. Although sophisticated offline AI models have been shown to perform well, they entail high inference costs and memory consumption. To overcome such drawbacks, a lightweight online algorithm is proposed to predict timeouts with high accuracy but without the heavy overhead.

The proposed heuristic is implemented in the Theta model checking framework, extending its existing portfolio capabilities to allow an algorithm to be stopped based on the predictive heuristic and to share partial results between algorithms. To evaluate the approach, an experiment was designed and executed on the benchmark set of the International Software Verification Competition (SV-COMP), analyzing how effective the proposed heuristic is in practice. The evaluation demonstrates that the simple partial result transfer method can solve previously intractable problems. Furthermore, the lightweight online heuristic predicts algorithm timeouts with high accuracy, proving to be a viable alternative to resource intensive offline models.

Chapter 1

Introduction

Every day, more and more software is written, and with it, the number of potential bugs continues to grow. Software verification is the crucial process of finding these bugs or proving that a program is free of them. For critical software, such as in aviation or medical devices, a failure can lead to huge losses, so a higher level of certainty is required. This is achieved through formal methods, which use mathematical tools to reason about software [29, 20]. A key formal method is model checking, where a tool explores all possible states of a program to check if it violates a given safety property [25, 29, 20].

The biggest challenge in model checking is the state space explosion. The number of states a program can be in can grow exponentially fast. For example, a program with just 20 boolean variables already has 2^{20} (around a million) possible combinations of states. Because of this, a major focus of research is finding efficient ways to explore this state space. Over the years, many different algorithms have been created, such as CEGAR [19, 22] (Counterexample Guided Abstraction Refinement), BMC [18] (Bounded Model Checking), IMC [26, 16] (Interpolation Based Model Checking), and k-Induction [1, 21] (KInd). Each of these has unique strengths and weaknesses. One might be good at handling loops, while another is better with complex data. As a result, no single model checking algorithm is the best for every situation.

Since different algorithms are good at solving different problems, a common strategy is to combine them in what is called a portfolio [4, 7]. A powerful way for these algorithms to work together is through cooperation, where they share information and partial results [3, 8, 12] . For instance, in a sequential portfolio, one algorithm runs for a while, and if it gets stuck, it can pass what it has learned to the next algorithm in line. However, many current methods for cooperation are very complex. Techniques like Conditional Model Checking [11], which uses an assumption automaton to pass information, or code reducing tools [13, 15] are powerful but have a high implementation barrier, making them difficult to use in practice.

To solve this, the first part of my contribution is a much simpler way to transfer partial results. My method extracts invariants from a CEGAR run that has been stopped. I then inject these invariants directly into the program's Control-Flow Automaton (CFA) [9, 29]. This gives the next algorithm a head start by providing it with proven facts about the program's behavior, and my approach is straightforward to implement.

However, sharing results is only half the problem. I also need to know the best time to stop the first algorithm. If I stop it too early, it might not have found any useful information. If I let it run for too long, valuable time is wasted. Most current approaches simply use a static time limit, which is not an adaptive solution. While some have tried using offline

AI models [28] to predict timeouts, these models can be slow at runtime and consume a lot of memory.

To address this, the second part of my contribution is a lightweight, online algorithm that predicts timeouts as the checker runs. I use a one dimensional Recursive Least Squares (RLS) [24] model to do this. The model observes the time taken for each iteration of the CEGAR algorithm and learns the performance pattern. Since the runtime often grows exponentially, my RLS model can accurately predict if the next iteration will take too long and exceed the time budget, allowing the algorithm to be stopped at just the right moment.

I implement my heuristic in the Theta model checking framework, which already contains the algorithms mentioned above. I extend its portfolio capabilities to allow for stopping based on my predictive heuristic and for sharing results using my simple injection method.

Finally, to prove that my method works, I design and run an experiment on the benchmark set from the International Software Verification Competition (SV-COMP) [6]. The evaluation results confirm the effectiveness of both contributions. The simplified partial result transfer mechanism, despite its simplicity, successfully solved 13 verification tasks that were previously unsolvable by the standalone algorithms, demonstrating its value on difficult problems. Furthermore, the online timeout prediction heuristic performed exceptionally well, achieving an F1 score of 0.77. The heuristic showed high reliability with a precision of 91.8% and a specificity of 95.7%, correctly identifying timeouts and non timeouts in the vast majority of cases. A lower recall of 66.5% was observed, which is an understood consequence of the current end of iteration update strategy. Overall, the evaluation confirms that the proposed methods are practical and effective enhancements for cooperative algorithm portfolios.

Chapter 2

Background

2.1 Model Checking

Model checking is a formal verification technique that systematically explores all possible *states* and *transitions* (the state space) of a *formal model* of a system to determine if it satisfies a formally specified property [20, 29, 25].

A simple and fundamental usage off model checking is in *Reachability Analysis*. In other words, given a *safety property*, the goal is to determine if it is possible to reach any undesirable states where this property is violated. Algorithm 1 presents a general model checking algorithm for reachability.

Algorithm 1 Model Checking for Safety Property (Reachability) P

```
1: Define $E$ := the set of states that violate property $P$ (the error states).
2: Let $Q$ := the set of initial states.
3: Let $\mathrm{Tran}(S)$ := the set of states reachable in one step from any state in a set $S$.
4: Let $Q_{old} \leftarrow \emptyset$
5: while $Q \neq Q_{old}$ and $Q \cap E = \emptyset$ do
6:     $Q_{old} \leftarrow Q$
7:     $Q \leftarrow Q \cup \mathrm{Tran}(Q)$            ▷ Expand $Q$ with newly reached states
8: end while
9: if $Q \cap E \neq \emptyset$ then
10:     return "Unsafe: Property $P$ is violated."
11: else
12:     return "Safe: Property $P$ holds."
13: end if
```

2.2 CFA and XCFA

To perform model checking, a program written in source code must first be transformed into a *formal model* of the chosing. One of a prominent formal models is *Control Flow Automaton* (CFA).

Definition 1 (Control flow automata [9]). Let:

- v := variable

- $D_v :=$ domain of variable v
- φ is an expression $\Rightarrow \text{var}(\varphi) = \{\forall v \mid v \in \varphi\}$
- predicate $:= p : D_{v_i} \times \cdots \times D_{v_n} \to \{\text{true}, \text{false}\}$

A control flow automaton is a tuple $\text{CFA} = (V, L, l_0, E)$ where:

- $V := \{v_1, \ldots, v_n \mid \text{domain}(v_i) = D_{v_i}\}$
- $L :=$ {program locations modeling the program counter}
- $l_0 \in L :=$ initial program location
- $E \subseteq L \times \text{Ops} \times L :=$ {directed edges representing operations}
- $\text{Ops} = \{\text{op} \mid \text{op} = \text{assignment} \vee \text{op} = \text{assumption}\}$
  - assignment $:= (v := \varphi)$ s.t. $v \in V \wedge \varphi \in D_v \wedge \text{var}(\varphi) \subseteq V$
  - assumption $:= [\varphi]$ s.t. $\varphi :=$ predicate $\wedge\ \text{var}(\varphi) \subseteq V$ ▪

```c
int main() {
  unsigned char n = nondet();
  if (n == 0) return 0;

  unsigned char v = 0;
  unsigned char s = 0;
  unsigned int i = 0;
  while (i < n) {
    v = nondet();
    s += v;
    ++i;
  }

  if (s < v) {
    reach_error();
    return 1;
  }

  return 0;
}
```

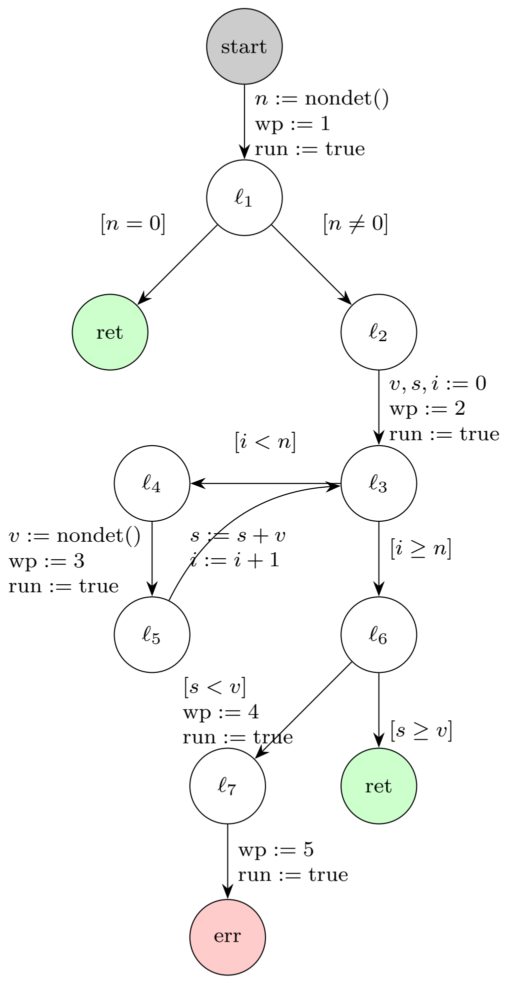


Figure 2.1: The unsafe C program on the left and its corresponding XCFA on the right.

Evidently, a Control Flow Automaton (CFA) is a formalism based on graphs [29]. Furthermore, *assignment* and *assumption* statements can be represented as *transition functions* (as shown below), making them applicable in model checking.

$$\begin{aligned}\mathrm{tran}(v := \varphi) &= (v' = \varphi) \wedge \bigwedge_{u \in V \setminus \{v\}} (u' = u),\\ \mathrm{tran}([\psi]) &= \psi \wedge \bigwedge_{v \in V} (v' = v).\end{aligned}$$

Although for the purpose of this report only the Control Flow Automaton (CFA) is needed, the actual implementation and evaluation of portfolios exchanging partial results in THETA employ an *Extended Control Flow Automaton* (XCFA). The XCFA is a version of a CFA designed to manage more intricate processes, such as multithreading. However, our current implementation does not utilize any of the additional capabilities offered by the XCFA. Figure 2.1 shows an example of a C program and its corresponding XCFA.

2.3 CEGAR

There are a diverse number of model checking algorithms. One of the most well-known in the literature is *Counterexample-Guided Abstraction Refinement (CEGAR)* [19, 22]. The CEGAR algorithm consists of two main phases that are executed iteratively.

- **Abstraction:** In the first phase, the *abstract reachability graph* (ARG) [10] is expanded by effectively exploring states until an error state is found in the abstract domain, which is defined by the current precision. If no error state is reachable, the program is deemed safe. Otherwise, we proceed to the second phase. A directed path between ARG nodes is called an *abstract trace*. If this path leads from the initial ARG node to an error state, it is called an *abstract counterexample*.
- **Refinement:** In the second phase, the refiner checks if the current *abstract counterexample* is feasible. Since the analysis operates in an abstract domain, we must check if the abstract path corresponds to a *concrete counterexample*. If the refiner confirms that the counterexample is concrete, the program is proven to be unsafe. Otherwise, the counterexample is considered *spurious*. In this case, we refine the precision and prune the ARG to prepare it for the next Abstraction phase.

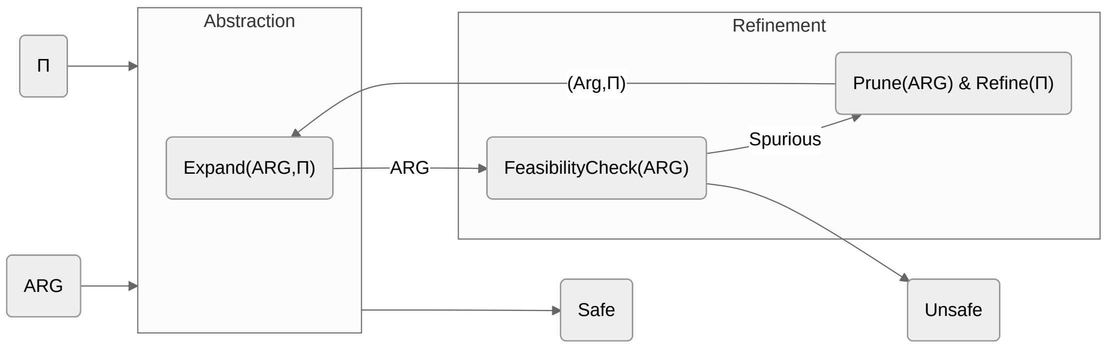


Figure 2.2: Schema of a Counterexample-Guided Abstraction Refinement (CEGAR) algorithm using ARG

The idea behind CEGAR is that the abstract domain resulting from a coarser precision is smaller and can be searched faster. However, this may lead to an abstract counterexample being *spurious* (i.e., not corresponding to a concrete execution path). Therefore, we need to check this counterexample during the refinement phase. The process continues by refining the precision, which makes the abstract state space larger and closer to the concrete state space. The CEGAR loop is run until the abstraction step proves safety, the precision cannot be refined any further, or a concrete counterexample is found. A general scheme of the CEGAR algorithm can be seen in Figure 2.2.

2.3.1 Abstraction

The Abstraction phase in CEGAR requires an ARG and a precision, which is continuously refined by the refiner. To construct an ARG, we need an *Abstract Domain* D and a *Transfer Function* T [19, 22]. The transfer function utilizes a *Precision* Π [19, 22] to compute the successor states.

Definition 2 (Abstract Domain [19]). An abstract domain is a tuple $D = (S, \top, \bot, \sqsubseteq, \text{expr})$ where

- S is a (possibly infinite) lattice of abstract states,
- $\top \in S$ is the top element,
- $\bot \in S$ is the bottom element,
- $\sqsubseteq \subseteq S \times S$ is a partial order conforming to the lattice, and
- $\text{expr} : S \mapsto \text{FOL}$ is the expression function that maps an abstract state to its meaning (the concrete data states it represents) using a First Order Logic (FOL) formula. ∎

Definition 3 (Precision [19]). A precision π is an element of the set of precisions Π. This set forms a partially ordered set $(\Pi, \preceq)$, where for any $\pi_1, \pi_2 \in \Pi$, we say π_1 is *more precise* than π_2, denoted $\pi_1 \preceq \pi_2$, if and only if for all $s \in S$ and all operations Ops:

$$T(s, \text{Ops}, \pi_1) \sqsubseteq T(s, \text{Ops}, \pi_2)$$

Definition 4 (Transfer Function [19]). Let S be a set of Abstract States, Π be a set of precisions, and Ops be a set of operations. Then the transfer function is $T : S \times \text{Ops} \times \Pi \mapsto 2^S$. ∎

In other words, T is a function that takes an abstract state, a precision, and an operation as input, and it computes the set of successor states which an be reached by applying the operation with the given precision.

This definition of abstraction is very general, and several types of abstraction can be derived from it. To specify a particular abstraction, we must define its Abstract Domain D, Transfer Function T, and Precision Π.

Boolean Predicate Abstraction

$D = (\langle \Pi, \wedge, \vee, \neg \rangle,\ \top = \textbf{true},\ \bot = \textbf{false},\ s_1 \sqsubseteq s_2 \Leftrightarrow s_1 \Rightarrow s_2,\ \text{expr}(s) = s)$

$\Pi = \{\pi \mid \pi \in \text{FOL predicates}\}$

$$T(s, op, \pi) = \bigvee \left\{ \bigwedge_{p_i \in \pi, \alpha(v_i)=\text{true}} p_i \wedge \bigwedge_{p_j \in \pi, \alpha(v_j)=\text{false}} \neg p_j \;\middle|\; \alpha \models (s \wedge op \wedge \bigwedge_{p_k \in \pi} (v_k \Leftrightarrow p'_k)),\ p_k \xrightarrow{op} p'_k \right\}$$

You can also give the transition fucntion like this:

$$T(s, op, \pi) = \bigvee \left\{ c \;\middle|\; c \in C,\ C = \left\{ \bigwedge_{i=1}^{|\pi|} l_i \;\middle|\; l_i \in \{\pi_i, \neg\pi_i\},\ \forall \pi_i \in \pi \right\},\ \exists\alpha : \alpha \models s \wedge op \wedge c',\ c \xrightarrow{op} c' \right\}$$

The intuition behind *Boolean Predicate Abstraction* [10, 22] is its high expressive power. This is because predicates can be combined using conjunction, disjunction, or negation in any form to represent the current state. The next state in boolean predicate abstraction is the disjunction of all conjunctive clauses (cubes) that can be formed from the predicates in the precision. A clause is included if there exists a reachable concrete state that transitions via the operation *op* from a state satisfying the current abstract state s into a state satisfying the transformed clause c'. In other words, the core idea is checking the entailment $\exists\alpha : \alpha \models s \wedge op \wedge c'$, which is the foundation of boolean predicate abstraction.

Cartesian Predicate Abstraction

$D = (\langle \Pi \cup \{\neg p \mid p \in \Pi\}, \wedge\rangle,\ \top = \textbf{true},\ \bot = p \wedge \neg p,\ s_1 \sqsubseteq s_2 \Leftrightarrow s_1 \Rightarrow s_2,\ \text{expr}(s) = s)$

$\Pi = \{\pi \mid \pi \in \text{FOL predicates}\}$

$$T(s, op, \pi) = \bigwedge\{p_i \mid p_i \in \pi,\ s \wedge op \Rightarrow p'_i\} \wedge \bigwedge\{\neg p_j \mid p_j \in \pi,\ s \wedge op \Rightarrow \neg p'_j\}$$

In contrast to Boolean Predicate Abstraction, a state in *Cartesian Predicate Abstraction* [10, 22] is nothing more than a conjunction of predicates or their negations from the precision set. The next state is easily calculated by checking for each predicate in the precision whether it holds or not after applying the operation from the previous state s. Because the state representation is constrained to only using conjunctions, Cartesian predicate abstraction has lower expressive power than boolean predicate abstraction.

Explicit Value Abstraction

V is a set of program variables.

D_v is the domain of variable v.

$D = \{s \mid s : V \mapsto (D_v \cup \{\top_{d_v}, \bot_{d_v}\}), \top(v) = \top_{d_v} \forall v,\ \bot \Leftrightarrow \exists v \text{ s.t } s(v) = \bot_{d_v}\quad s_1 \sqsubseteq s_2 \Leftrightarrow \forall v,\ s_1(v) = s_2(v) \vee s_1(v) = \bot_{d_v} \vee s_2(v) = \top_{d_v},\ \text{expr}(s)\}$

$$\text{expr}(s) = \begin{cases} \textbf{false} & \text{if } s = \bot \\ \bigwedge_{v,\, s(v) \neq \top_{d_v}} (v = s(v)) & \text{otherwise} \end{cases}$$

$\Pi \subseteq V$

$$T(s, op, \pi) = s', \quad s'(v) = \begin{cases} v \xrightarrow{op} v' & v \in \pi \\ \top_{d_v} & \text{otherwise} \end{cases}$$

Explicit Value Abstraction [10, 22] is the simplest one by far. It operates by assigning concrete values to variables and keeping track of these assignments in the program.

As it can be seen Expli Value Abstaction only uses the equal operation to represent the states while Carteion Predicate negatoin and the and operta (which usin this you can make the equal x>0 and not x>2), while the boolean predicate also has the power of or. The expresoin power goes up but the computatoin cost for the transition function also goes up.

2.3.2 Abstraction with Locatoins

In the context of the XCFA, it is essential to track not only the abstract state (s) but also the program locations (l). This is necessary because the primary objective is to determine whether an error at a specific location is ever reachable. The fundamental approach involves traversing the XCFA to find an error location sequence, such as $l_0, l_1, \ldots, l_{\text{error}}$, if the error location exists. However, this method is insufficient because it does not account for the fact that some conditional branches (e.g., an `if` branch) may never be traversed by the program because the branch condition is never fulfilled. Therefore, to determine the feasibility of an `if` branch's execution, we must maintain the state of the program, specifically the range of values that can be assigned to the variables. This can be achieved by using either Explicit States or Predicate States. For instance, if the current abstract state implies that $x > 10$, then the branch conditioned on $x \leq 10$ will not be traversed, and consequently, any error location within that branch will never be reached.

In other words, we are trying to check for the existence of an abstract path P:

$$P = (l_0, s_0) \xrightarrow{\text{op}_0} (l_1, s_1) \xrightarrow{\text{op}_1} \ldots \xrightarrow{\text{op}_{n1}} (l_n, s_n) \xrightarrow{\text{op}_n} (l_{\text{error}}, s_{n+1})$$

This path relies on two combined concepts:

1. The operations (op_i) located on the XCFA edges, which are applied to the abstract states.
2. The edges themselves, which define the transition to the next location.

Putting these two concepts together yields an extended transfer function $T_{\mathcal{L}}$. A pair $s_{\mathcal{L}} = (l, s)$ represents a state in this extended abstract domain, $D_{\mathcal{L}}$.

Now lests precicely define D_l and T_l.

Definition 5 (Abstract Domain with Locations [22]). Let $\mathcal{L}$ be the set of locations from a Formal Model, and let S be the set of abstract states from the base abstract domain D. The abstract domain D extended with locations is a tuple $D_{\mathcal{L}} = (S_{\mathcal{L}}, \top_{\mathcal{L}}, \bot_{\mathcal{L}}, \sqsubseteq_{\mathcal{L}}, \text{expr}_{\mathcal{L}})$ where:

- $S_{\mathcal{L}} = \mathcal{L} \times S$, which is the set of abstract states paired with a location.
- $\top_{\mathcal{L}} = \{(l, \top) \mid l \in \mathcal{L}\}$, which is the set of top elements, one for each location.
- $\bot_{\mathcal{L}} = \{(l, \bot) \mid l \in \mathcal{L}\}$, which is the set of bottom elements, one for each location.
- The partial order $\sqsubseteq_{\mathcal{L}}$ is defined such that for any two extended abstract states $s_{1\mathcal{L}} = (l_1, s_1)$ and $s_{2\mathcal{L}} = (l_2, s_2)$, we have $s_{1\mathcal{L}} \sqsubseteq_{\mathcal{L}} s_{2\mathcal{L}}$ if and only if $l_1 = l_2$ and $s_1 \sqsubseteq s_2$.
- $\text{expr}_{\mathcal{L}} \equiv \text{expr}$, where expr is the expression function of the base abstract domain. ▪

Before defining the extended transfer function $T_{\mathcal{L}}$, we first introduce the concept of location dependent precision, $\Pi_{\mathcal{L}}$.

Definition 6 (Precision with Locations [22]). Let $\mathcal{L}$ be the set of locations from a formal model. Then $\Pi_{\mathcal{L}} = \{(l, \Pi) \mid l \in \mathcal{L} \text{ and } \Pi \text{ is a precision}\}$. ▪

When each location can be mapped to a potentially different precision, this is known as **local precision**. If the same precision is used for all locations, this is knows as **global precision**. It follows that the global precision strategy is merely a special case of the more general local precision approach.

Definition 7 (Transfer Function with Locatoins [22]). Let $\mathcal{L}$ be the set of locations from an Formal Model, S be a set ob Abstract States, $\Pi_{\mathcal{L}}$ be a precision dependin in location and Ops a set of operations, then the transfer function with Locatoins is $T_{\mathcal{L}} : S_{\mathcal{L}} \times \text{Ops} \times \Pi_{\mathcal{L}} \rightarrow 2^{S_{\mathcal{L}}}$. ▪

An Abstraction with Locations is an Abstraction in itself which can be conbined with other abstractions. For expmaple you can keep track of locations and also have Explicit Value abstractoin or Cartesian Predicatge Abstraction.

2.3.3 ARG

The Abstract Reachability Graph (ARG) is a core component of the CEGAR algorithm. After reviewing Abstraction in Section 2.3, we are now ready to define the ARG precisely.

Definition 8 (Abstract Reachability Graph [10]). Let an abstraction be defined by an Abstract Domain $D = (S, \top, \bot, \sqsubseteq, \text{expr})$, a precision Π, and a transfer function T. An Abstract Reachability Graph (ARG) is a tuple $ARG = (N, E, C)$, where:

- $N \subseteq S$ is the set of *nodes*.
- $E \subseteq N \times \text{Ops} \times N$ is the set of *directed edges*. An edge $(s_1, \text{op}, s_2) \in E$ exists if $s_2 \in T(s_1, \text{op}, \Pi)$.
- $C \subseteq N \times N$ is the set of *coverage edges*. An edge $(s_1, s_2) \in C$ exists if $s_1 \sqsubseteq s_2$. ▪

The role of nodes and directed edges is straightforward, but the reason for including *coverage edges* requires explanation. The purpose of coverage is to prune the search space. If, during the exploration of the state space, a newly generated state s_{new} is found to be covered by an existing state s_{old} in the graph (i.e., $s_{new} \sqsubseteq s_{old}$), then further exploration from s_{new} can be halted. This is because any path originating from the more *specific state* s_{new} is implicitly accounted for by the exploration of the more *general state* s_{old}.

2.4 BMC and K-Induction

Bounded Model Checking (BMC) [18] is a widely implemented algorithm in today's verification landscape. It translates the verification problem into a formula that can be solved by various methods, including Binary Decision Diagrams (BDDs) or *SAT/SMT* [18] solvers. For the purposes of this report, we are interested in BMC and *K-Induction* [1, 21], specifically when using SAT/SMT solvers.

First, let us define a Transition System.

Definition 9 (Transition System [18]). A transition system M is a tuple $M = (S, I, T)$ where:

1. S is the set of states, representing all the configurations the system can be in.
2. $I \subseteq S$ is the set of initial states.
3. $T \subseteq S \times S$ is the transition relation. If $(s, s') \in T$, it implies that the system can transition from state s to s'. ▪

In our context, the *system* is the program under analysis. The *states* are represented as First-Order Logic (FOL) expressions over program variables, combined with the locations in the program.

Definition 10 (BMC Query [18]). Let $M = (S, I, T)$ be a Transition System. Let $\mathbf{P}(s)$ be a formula representing all states satisfying a safety property, and let $k \in \mathbb{Z}^+$ be the time bound.

To perform Bounded Model Checking (BMC) using a SAT/SMT solver, we construct the Bounded Safety Formula $\mathbf{\Phi}_k$ as follows:

$$\mathbf{\Phi}_k = \mathbf{I}(s_1) \wedge \bigwedge_{i=1}^{k-1} \mathbf{T}(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \overline{\mathbf{P}(s_i)}$$

If $\mathbf{\Phi}_k$ is satisfied, it means that there exists a path of length k or less that violates the safety property $\mathbf{P}$. Otherwise, the program is safe up to the bound k. To put it simply, BMC checks all execution paths of length k or less to see if any of them lead to an error. It is clear that BMC is only complete up to the given bound k, a property often referred to as *bounded completeness.*

For a concrete example, let's suppose we have a CFA, and we are keeping track of the concrete state (explicit value states for all variables and the current location). Our transition system is defined as: $M = (S, I, T)$ where $S = L \times \{s \mid s : V \to D_v \cup \{\top_{dv}, \bot_{dv}\}\}$, $I = \{(l_{\text{start}}, s_{\text{init}}) \mid s_{\text{init}}(v) = \top_{dv}\}$, $T = \{((l, s), (l', s')) \in S \times S \mid \exists (l, op, l') \in E_{CFA} \wedge s' \neq \bot\}$

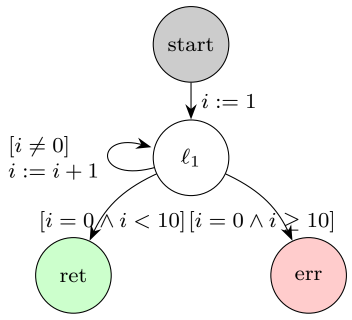
Let's choose a bound of $k = 3$ to unroll the CFA. The first transition will be taken by the statement `uchar i=1`, so we will have two transitions left to unroll the loop. Figure 2.3 shows the original CFA on the left and the unrolled CFA on the right. The property we want to check is $P(S = (l, s)) = (l \neq l_{\text{err}})$.

The transition relation T is defined in such a way that transitions leading to a bottom state ($\bot$) are pruned. For example, any transition from l_1 to l_2 where the assumption $i \neq 0$ is not met would be excluded. This means the transition relation effectively checks if the current state satisfies the assumptions on an edge; if not, that transition is not included in the relation.

The BMC query is:

$$
\begin{aligned}
\mathbf{\Phi}_4 = &\underbrace{(\mathrm{loc}_1 := \ell_1 \wedge i_1 := 1)}_{\mathbf{I}(s_1)} \\
&\wedge \underbrace{\overbrace{(\mathrm{loc}_1 = \ell_1 \wedge i_1 \neq 0)}^{\text{Assumption}} \wedge \overbrace{(\mathrm{loc}_2 := \ell_1 \wedge i_2 := i_1 + 1)}^{\text{Assignment}}}_{\mathbf{T}(s_1,s_2)\ (\text{Loop } 1)} \\
&\wedge \underbrace{(\mathrm{loc}_2 = \ell_1 \wedge i_2 \neq 0) \wedge (\mathrm{loc}_3 := \ell_1 \wedge i_3 := i_2 + 1)}_{\mathbf{T}(s_2,s_3)\ (\text{Loop } 2)} \\
&\wedge \Big(\underbrace{(\mathrm{loc}_1 = \mathrm{err})}_{\overline{\mathbf{P}(s_1)}} \vee \underbrace{(\mathrm{loc}_2 = \mathrm{err})}_{\overline{\mathbf{P}(s_2)}} \vee \underbrace{(\mathrm{loc}_3 = \mathrm{err})}_{\overline{\mathbf{P}(s_3)}} \Big)
\end{aligned}
$$

```
uchar i=1;
while (i) {
  i++;
}
assert(i<10);
```



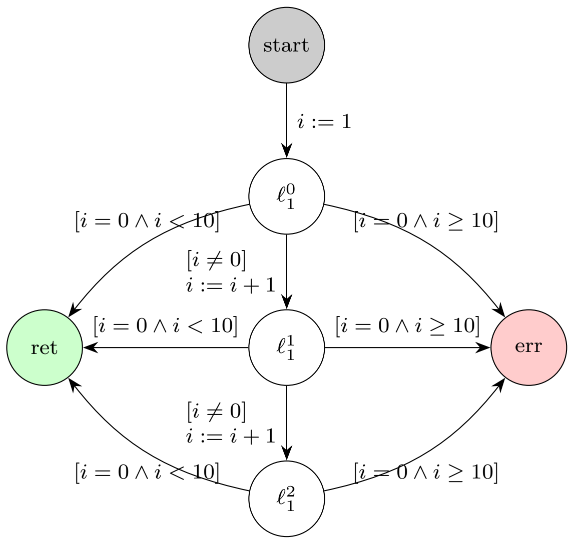


Figure 2.3: A simple `while loop` and its CFA (left). Unrolled CFA with a bound of $k = 3$ showing all possible exits (right).

The BMC would conclude that the program is correct because our bound is only $k = 4$. To find the error, BMC would require a bound of at least $k = 11$, since the loop must be unrolled 10 times to violate the assertion ‘i<10‘.

To address the *incompleteness* of BMC, several solutions have been proposed, such as K-Induction and IMC. K-Induction is an extension of BMC. The classic Split-Case K-Induction algorithm is performed in two steps: the *Base Case*, which is a standard BMC query with a bound k, and the *Step Case*, where induction is performed.

Definition 11 (K-Induction Queries [21]). Let $M = (S, I, T)$ be a Transition System, let $\mathbf{P}(s)$ be a formula representing the safety property, and let $k \in \mathbb{Z}^+$ be the bound. K-Induction is performed in two steps:

1. **Base Step:** Check that no paths of length k or less lead to an error. This is done by checking if the following BMC formula is **unsatisfiable**.

$$\mathbf{\Phi}_k = \mathbf{I}(s_1) \wedge \bigwedge_{i=1}^{k-1} \mathbf{T}(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \overline{\mathbf{P}(s_i)}$$

2. **Inductive Step:** Check if the property is inductive for any path of length k. This can be done by checking that the following formula is **unsatisfiable**.

$$\mathbf{\Psi}_k = \left(\bigwedge_{i=1}^{k} \mathbf{T}(s_i, s_{i+1}) \right) \land \left(\bigwedge_{i=1}^{k} \mathbf{P}(s_i) \right) \land \overline{\mathbf{P}(s_{k+1})}$$

If the Base Step formula is unsatisfiable up to bound k and the Inductive Step formula is also unsatisfiable for the same bound k, the property $\mathbf{P}$ is proven to be an invariant of the system. ▪

To better explain the Inductive Step, the formula can be rearranged as follows:

$$\mathbf{\Psi}_k = \underbrace{\mathbf{P}(s_1) \land \cdots \land \mathbf{P}(s_k) \land \bigwedge_{i=1}^{k-1} \mathbf{T}(s_i, s_i + 1) \land}_{\text{Part 1: The Hypothesis}} \quad \underbrace{\mathbf{T}(s_k, s_{k+1})}_{\text{Part 2: The Next Step}} \quad \land \quad \underbrace{\overline{\mathbf{P}(s_{k+1})}}_{\text{Part 3: The Contradiction}}$$

The intuition behind K-Induction is that a larger bound k provides more *context.* This results in a stronger set of *assumptions* for the hypothesis, which can make it easier to prove the inductive step.

For the example in Figure 2.3, the Inductive Step query would be as follows:

$$\mathbf{\Psi}_k = \underbrace{(\text{loc}_1 \neq \ell_{\text{err}})}_{\mathbf{P}(s_1)} \land \mathbf{T}(s_1, s_2) \land \cdots \land \mathbf{T}(s_{k-1}, s_k) \land \underbrace{(\text{loc}_k \neq \ell_{\text{err}})}_{\mathbf{P}(s_k)}$$

$$\land \underbrace{\begin{pmatrix} \text{loc}_k = \ell_1 \land i_k \neq 0 \land \text{loc}_{k+1} = \ell_1 \land i_{k+1} = i_k + 1 \\ \lor \ \text{loc}_k = \ell_1 \land i_k = 0 \land i_k < 10 \land \text{loc}_{k+1} = \ell_{\text{ret}} \\ \lor \ \underbrace{\text{loc}_k = \ell_1 \land i_k = 0 \land i_k \geq 10 \land \text{loc}_{k+1} = \ell_{\text{err}}}_{\text{evaluates to false}} \end{pmatrix}}_{\mathbf{T}(s_k, s_{k+1})}$$

$$\land \underbrace{(\text{loc}_{k+1} = \ell_{\text{err}})}_{\overline{\mathbf{P}(s_{k+1})}}$$

In the K-Induction step, the initial state formula $I(s_1)$ is omitted. Consequently, we cannot make any assumptions about the starting values of the state variables, such as i_1.

By not including the initial state, we shift the focus from properties of specific paths to general properties of the system's transitions; we are searching for *invariants.* The query $\mathbf{\Psi}_k$ is unsatisfiable because the only transition that leads to the error location requires the condition $(i_k = 0 \land i_k \geq 10)$. This conjunction is a logical contradiction. Since reaching $\text{loc}_{k+1} = \ell_{\text{err}}$ is only possible via a transition that contains this logical contradiction, the entire formula is unsatisfiable. Assuming the base case also holds, we can therefore conclude that the program is safe.

2.5 Portfolios

No single algorithm is best for verifying all programs, as each possesses its own strengths and weaknesses. Consequently, to exploit these individual strengths, different algorithms are often combined into *portfolios* [4, 7]. A portfolio is typically a *black box model*, meaning that the individual algorithms do not need to be aware of one another to operate within the framework. Essentially, algorithms are selected and arranged to run either in *parallel* or *sequentially* [4, 7].

An important feature of a portfolio is its ability to handle failures. If one algorithm (or batch of algorithms) fails, the portfolio can decide which one to execute next. For example, if an explicit state model checker encounters a state space explosion, it might be beneficial to switch to predicate abstraction. Similarly, if a solver error occurs, it could be a good strategy to try K-Induction.

In the literature, it is common to run several algorithms in parallel batches, and then execute these batches sequentially. Figure 2.4 shows a generic portfolio structure.

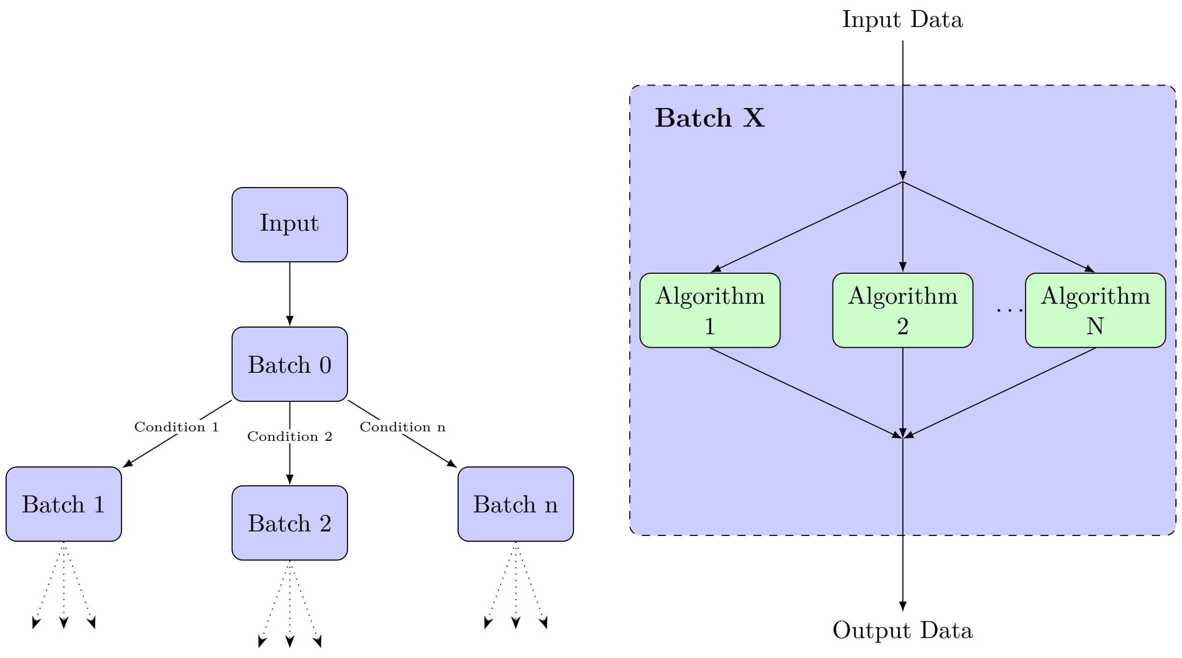


Figure 2.4: A generic portfolio structure (left) and the internal structure of a generic batch (right).

In this report, we will focus exclusively on sequential portfolios. Figure 2.5 illustrates a concrete, purely sequential portfolio where the next algorithm is chosen if any error occurs in the current one. In other words, if an exception is thrown, the portfolio proceeds to the next configured algorithm.

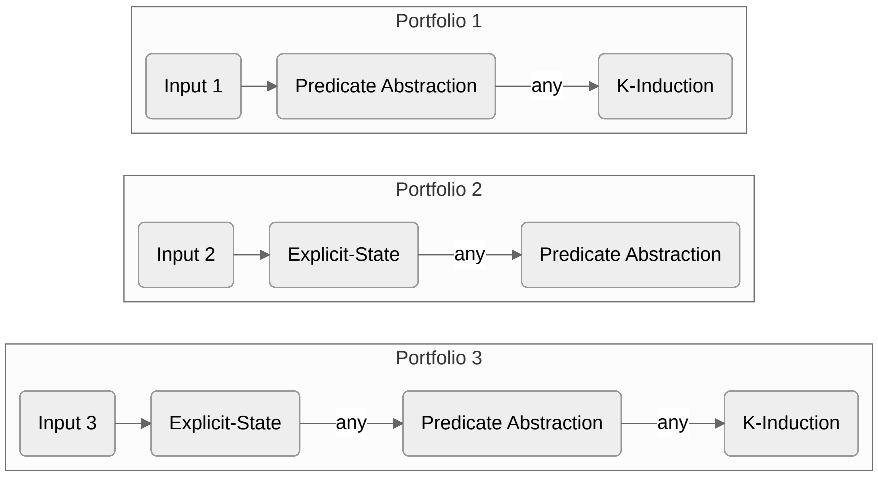


Figure 2.5: Examples of concrete sequential portfolios. One portfolio switches from Predicate Abstraction to K-Induction, while another switches from Explicit Value Abstraction to K-Induction upon failure.

A portfolio terminates as soon as any of its algorithms successfully verifies the program. It fails only if all configured algorithms fail to do so.

2.6 One Dimensional Recursive Least Squares Predictor

Suppose we have a set of data points, $(x_1, y_1), \ldots, (x_t, y_t)$, collected sequentially up to the current time t. For each point, x_i is the input and y_i is the corresponding observed output. We assume a simple linear model with a single parameter, w, which we want to learn by minimizing the sum of squared errors. This is known as finding the *Least Squares* solution.

To allow the model to adapt to new data over time, we introduce an exponential forgetting factor, λ. The corresponding cost function, $J(t)$, is given by:

$$J(t) = \sum_{i=1}^{t} \lambda^{t-i} (y_i - w(t)x_i)^2$$

The weight, $w(t)$, is dependent on time because a new data point at $t+1$ requires a recalculation of the optimal parameter. The forgetting factor, λ, ensures that more recent data points have a greater influence on the solution.

To find the optimal weight $w(t)$ that minimizes $J(t)$, we take the derivative with respect to $w(t)$ and set it to zero.

$$\frac{dJ(t)}{dw(t)} = \sum_{i=1}^{t} \lambda^{t-i} \cdot 2(y_i - w(t)x_i)(-x_i) = 0$$

We can simplify this expression by dividing by -2 and rearranging the terms.

$$\sum_{i=1}^{t} \lambda^{t-i} (y_i - w(t)x_i)(x_i) = 0$$

Distributing x_i and separating the summation yields:

$$\sum_{i=1}^{t} \lambda^{t-i} y_i x_i - \sum_{i=1}^{t} \lambda^{t-i} w(t) x_i^2 = 0$$

Since $w(t)$ is constant with respect to the summation index i, we can factor it out:

$$w(t) \sum_{i=1}^{t} \lambda^{t-i} x_i^2 = \sum_{i=1}^{t} \lambda^{t-i} y_i x_i$$

Solving for $w(t)$ gives us the final batch least-squares solution:

$$\boxed{w(t) = \frac{\sum_{i=1}^{t} \lambda^{t-i} y_i x_i}{\sum_{i=1}^{t} \lambda^{t-i} x_i^2}}$$

This formula represents the non-recursive, batch least-squares solution with a forgetting factor. It serves as the starting point for deriving the online, recursive equations.

2.6.1 The Online Solution

Calculating $w(t)$ from scratch every time a new data point (x_t, y_t) arrives is computationally inefficient. Instead, we can develop a method to update the weight incrementally. The goal is to find a function f such that $w(t) = f(w(t-1), (x_t, y_t))$.

To begin, we define the numerator and denominator of the batch solution as $q(t)$ and $R(t)$, respectively.

$$q(t) = \sum_{i=1}^{t} \lambda^{t-i} y_i x_i$$
$$R(t) = \sum_{i=1}^{t} \lambda^{t-i} x_i^2$$

These can be expressed in a recursive form:

$$q(t) = \lambda q(t-1) + y_t x_t$$
$$R(t) = \lambda R(t-1) + x_t^2$$

Now, we define $P(t) = 1/R(t)$. Using this definition and the recursive formula for $R(t)$, the Sherman-Morrison formula can be applied to find a recursive update for $P(t)$:

$$P(t) = \frac{1}{\lambda \frac{1}{P(t-1)} + x_t^2} = \frac{P(t-1)}{\lambda + P(t-1) x_t^2}$$

With these relationships, we can derive the update for $w(t) = P(t)q(t)$:

$$w(t) = P(t)[\lambda q(t-1) + y_t x_t]$$
$$w(t) = P(t)[\lambda R(t-1) w(t-1) + y_t x_t]$$
$$w(t) = P(t)\left[\frac{\lambda}{P(t-1)} w(t-1) + y_t x_t\right]$$

From the definition of $P(t)$, we know $\lambda P(t-1)^{-1} = P(t)^{-1} - x_t^2$. Substituting this gives:

$$\begin{aligned}
w(t) &= P(t)[(P(t)^{-1} - x_t^2)w(t-1) + y_t x_t] \\
w(t) &= w(t-1) - P(t)x_t^2 w(t-1) + P(t)y_t x_t \\
w(t) &= w(t-1) + P(t)x_t(y_t - x_t w(t-1))
\end{aligned}$$

By substituting the practical formula for the gain, we arrive at the complete online update rule:

$$\boxed{w(t) = w(t-1) + \frac{P(t-1)x_t}{\lambda + P(t-1)x_t^2}(y_t - x_t w(t-1))}$$

Let's analyze the components of this important equation.

The *prediction error* is:

$$e(t) = y_t - w(t-1)x_t$$

The *covariance update* is:

$$P(t) = \frac{P(t-1)}{\lambda + P(t-1)x_t^2}$$

The *gain term*, which scales the error, is given by:

$$k(t) = P(t)x_t = \frac{P(t-1)x_t}{\lambda + P(t-1)x_t^2}$$

A simplified representation of the weight update is given by the equation:

$$w(t) = w(t-1) + k(t)e(t)$$

The *covariance term*, $P(t)$, represents the uncertainty of our weight estimate. In other words, it reflects our confidence in the prediction. A higher covariance (greater uncertainty) leads to a larger gain, resulting in a more significant update to the weight. Conversely, a smaller covariance indicates more certainty, resulting in a smaller gain and a more subtle weight adjustment.

For these equations to work, we must initialize the model with starting values for the weight, $w(0)$, and the covariance, $P(0)$, and also specify the forgetting factor, λ.

- The initial weight, $w(0)$, can be chosen based on prior knowledge, but is often set to zero if no information is available.
- We must start with a high initial covariance, $P(0)$, to reflect high initial uncertainty.
- The forgetting factor, λ, is usually chosen to be between 0.9 and 1.0. A value of 1 means no forgetting, while a value like 0.9 represents a high forgetting rate, as the influence of older data points decays quickly.

These equations allow us to update the weights online as new data points arrive. Furthermore, while we have only considered a linear model with a single parameter, this framework can be extended to include a bias term and multiple parameters [24]. The derivation is analogous but involves vectors and matrices instead of scalar values [24].

2.6.2 Application to Exponential Functions via Linearization

Although the RLS algorithm is designed for linear relationships, it can be adapted for non-linear models through *linearization.* The relationship of interest in this report is the exponential function $y = x^w$. We want to approximate the parameter w given a stream of data points. To achieve this, we can use the RLS predictor by first scaling our data points with the natural logarithm, which linearizes the relationship.

$$\begin{aligned} \ln(y) &= \ln(x^w) \\ \ln(y) &= w \cdot \ln(x) \end{aligned}$$

If we define new variables, $y' = \ln(y)$ and $x' = \ln(x)$, the relationship becomes:

$$y' = w \cdot x'$$

This transformed equation represents a simple line through the origin, which is exactly the model our one-dimensional RLS algorithm is designed to solve.

Chapter 3

Partial Results Transfer

In this chapter, we will introduce the concept of cooperation and explain some of the ways it can be implemented. We will then present the problem statement and proceed to detail our proposed solutions, explaining how to transfer partial results between algorithms. This will be done by describing how the partial results are extracted and subsequently injected. Finally, we will propose a more intelligent approach for algorithm termination than a simple time limit.

3.1 Cooperation

As mentioned, no single algorithm can solve all the specifics of every program. One approach to this problem is sharing information about the execution of one algorithm with another, a strategy known as *Cooperation* [4, 7]. This shared information, which constitutes a *partial result*, can take various forms. It can be a specific automaton, such as a *General Intermediate Automaton (GIA)* [23] or an *Assumption Automaton* [11] (which contains the state space that has been successfully verified by using assumptions), or it can be a *witness* [2, 14] that contains derived invariants. Furthermore, *Reducers* [13, 15] have become a very active research area in this field, specifically for passing these partial results to tools that do not natively support full cooperation. Another promising new method in this area is *Dynamic Program Splitting* [27].

A critical challenge in cooperative algorithm portfolios is determining the optimal runtime for an algorithm before extracting its partial result and passing it to a subsequent one. This is a challenging problem because it necessitates in depth knowledge of the algorithms and the ability to predict their future behavior. Solutions for predicting when an algorithm will not be able to finish its run, allowing for the extraction and transfer of partial results, have been proposed using pretrained AI models that incorporate dynamic and static program parameters [28]. However, this approach introduces several issues: the models tend to be overfitted to specific programs, querying the pretrained model must occur during runtime, and the pretrained model itself can consume significant memory.

3.2 Problem Statement

1. The concept of sequentially passing partial results within an algorithm portfolio to reduce the runtime of a subsequent algorithm is well documented in the literature. However, these methods are often complicated and implementation intensive. In

this report, we present a simpler approach: extracting invariants (similar to witness construction) from a partially executed CEGAR run and injecting them into the XCFA of the next algorithm.

2. While timeout prediction has been explored using pre trained, offline AI models, there has been no prior work on using lighter, online models for this purpose. In this paper, we propose an online prediction algorithm to forecast the timeout of a CEGAR based analysis using a Recursive Least Squares (RLS) algorithm.

3.3 Analyzing CEGAR and K-Induction

CEGAR is an algorithm that analyzes specific abstracted paths *globally*, from the initial state to a potential error state, while K-Induction is a more *local* algorithm that attempts to prove properties based on the transition relation itself.

A common problem for CEGAR is that proving safety requires it to traverse all paths in the ARG. When a high precision is necessary, this can be very slow due to the large number of states, a phenomenon known as *state space explosion.* This issue is demonstrated in Example B.1. Another potential problem is that the interpolation step during refinement may fail to infer sufficiently strong predicates.

Conversely, the main problem with K-Induction is that it only discovers invariants based on the properties of the transition relation. This means it does not make any assumptions regarding the initial state of the program's variables; in other words, K-Induction is *local.* This can be a significant drawback when the error condition only holds if the initial state is considered, which is something CEGAR handles by analyzing full paths from the start. This scenario is illustrated in Example B.2. Another problem that standard K-Induction faces is the need to unroll loops. For programs with deeply nested loops, this unrolling process can be very slow.

A detailed explanation of both examples, one where Predicate Abstraction succeeds while K-Induction fails, and another where K-Induction succeeds while Predicate Abstraction fails, can be found in Appendix B.

3.4 Partial Results Transfer

Now that we have an understanding of when one algorithm may outperform another, the question is how to effectively extract information from one and inject it into the other. There are various approaches to this transfer, which differ depending on the algorithms involved. We will specifically explain how to extract *location invariants* from an incomplete run of CEGAR and inject them into the XCFA.

Partial Results Extraction

Partial results are extracted from an incomplete CEGAR run by traversing the entire ARG that was constructed with the current precision. For each location, we then collect all the abstract states associated with it and combine them using a disjunction to form a location invariant.

Let $ARG = (N, E, C)$ be the Abstract Reachability Graph, where each node in N is a pair (l, s) of a location and an abstract state. Let $R(l) = \{s \mid (l, s) \in N\}$ be the set of all abstract states associated with a location l in the ARG. The location invariant for l, denoted $\mathcal{I}_l$, is the disjunction of all these states:

$$\mathcal{I}_l = \bigvee_{s \in R(l)} \mathrm{expr}(s)$$

The extracted location invariant is an *overapproximation* of the concrete program states that can be reached at that location.

Partial Results Injection

After obtaining the invariant $\mathcal{I}_l$ for each location, we inject these invariants into the XCFA as *assumptions*. The subsequent algorithm thus receives an XCFA augmented with this new information. The injection mechanism works as follows: for each location l with a corresponding invariant $\mathcal{I}_l$, we introduce a new intermediate location, let's call it $l_{partial}$. All incoming edges to the original location l are rerouted to this new location $l_{partial}$. Then, a single edge is created from $l_{partial}$ back to l, and this edge is labeled with the assumption $[\mathcal{I}_l]$. Figure 3.1 illustrates this process.

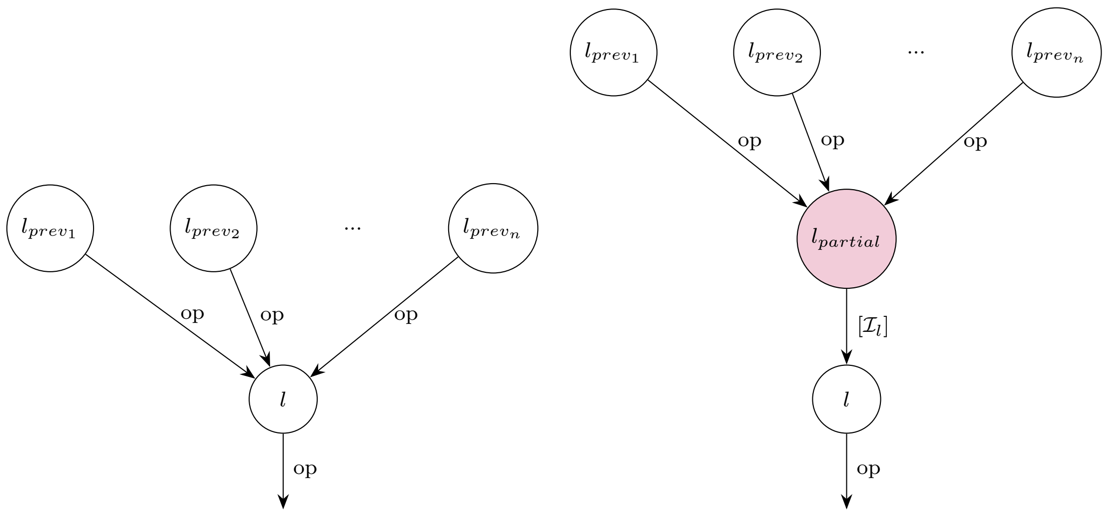


Figure 3.1: Left the XCFA of the program. Right the XCFA after partial results were injected.

3.4.1 Example of Transferring Partial Results between CEGAR and K-Induction

Below, we will explain the process for a portfolio of the type PredCart(pRes) $\xrightarrow{\text{any}}$ Kind().

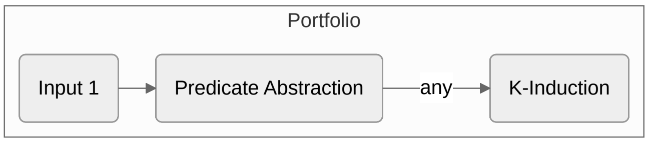


Figure 3.2: A portfolio of PredCart(pRes) $\xrightarrow{\text{any}}$ Kind().

```
int main() {
    int a = 2, b = 2;
    for(a = 0; a < 2; ++a) {
        for(b = 0; b < 2; ++b) {

        }
    }
    assert(a == 2 && b == 2);
}
```

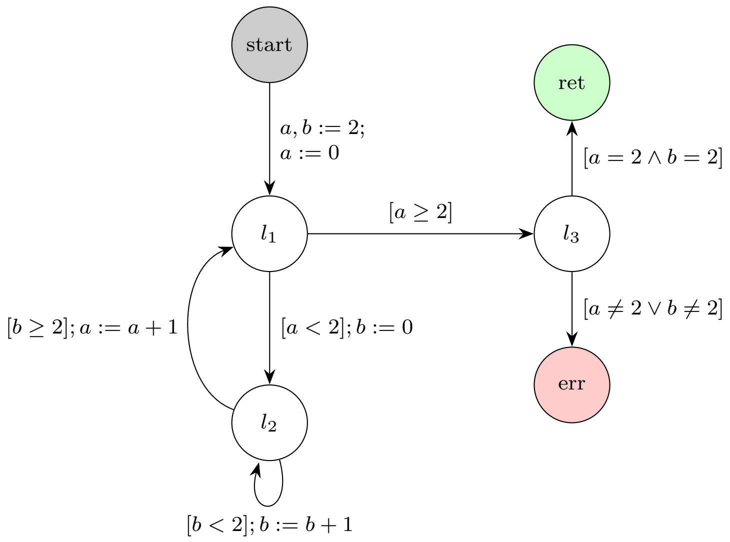


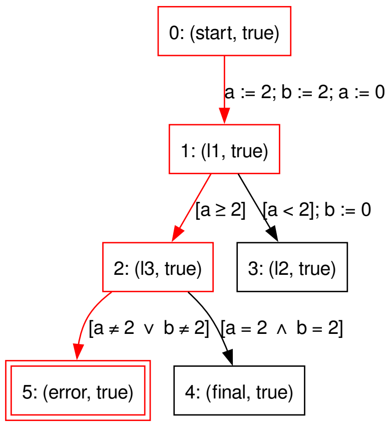
Figure 3.3: A CFA for the nested loop program with an explicit exit node before branching to the final assertion state.

The program in Figure 3.3 is difficult to verify for both a CEGAR based checker with predicate abstraction and for K-Induction. For Predicate Abstraction, this is mainly because the assertion requires the *exact values* of the variables to be tracked. In other words, the analysis would need to find predicates like $x = 2$ for each $x \in \{a, b\}$, which is problematic.

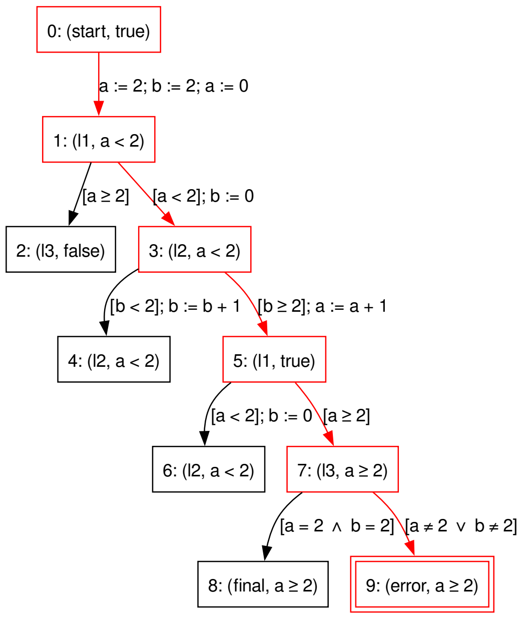
To explain why, let's analyze how CEGAR would attempt to verify this program. For the first iteration, the precision set Π is empty, which means the abstractor can only use the most general abstract states, $\top$ (true) and $\bot$ (false). After searching the graph, the abstractor finds a possible counterexample, highlighted in red in Figure 3.4a. The refiner determines that this counterexample is *spurious* because between node 0 and 1, the program executes 'a:=0', and then from node 1, the spurious path takes the branch conditioned on $a \geq 6$. The refiner encounters this first infeasible step and therefore updates the precision, for instance, with the predicate $a \geq 2$.

Next, the ARG is searched with the precision $\Pi = \{(a \geq 2)\}$. From this, the abstractor finds another counterexample, as shown in Figure 3.4b. The problem in this trace is between nodes 1, 3, and 5. When transitioning from node 1 to 3, the program sets 'b:=0', but the next step from 3 to 5 assumes $b \geq 2$, which is incorrect. Therefore, the refiner updates the precision to $\Pi = \{(a \geq 2), (b \geq 2)\}$. After unrolling with this new precision, we get the ARG shown in Figure 3.4c.

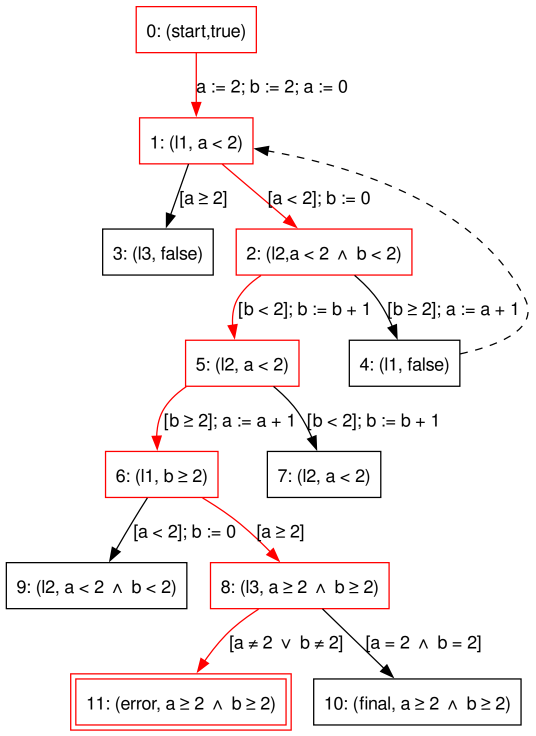
The refiner now identifies a new spurious path. It realizes that between nodes 5 and 6, the path is infeasible because it assumes $b \geq 2$, which is not true. From node 1 to 2, 'b' is set to 0, and then from node 2 to 5, 'b' is incremented to 1. Therefore, the refiner decides to track the value of 'b' more closely by adding $(b \leq 0)$ to the precision. For the next



(a) ARG with $\Pi = \{\}$



(b) ARG with $\Pi = \{(a \geq 2)\}$



(c) ARG with $\Pi = \{(a \geq 2), (b \geq 2)\}$

Figure 3.4: The expanded ARG with the given precision Π in each iteration of the CEGAR.

iteration, it will need to add $(b \leq 1)$, and then repeat the same process for 'a' by adding $(a \leq 0)$ and then $(a \leq 1)$. The precision eventually becomes:

$$\Pi = \{(a \geq 2), (b \geq 2), (b \leq 0), (b \leq 1), (a \leq 0), (a \leq 1)\}$$

The problem here is clear: PredCart is forced to keep track of the *exact values* of the variables because it cannot generalize. This makes the precision set very large and the solver queries slow. The size of the required precision is proportional to the number of variables and the loop bounds:

$$|\Pi| \approx (\text{Number of Variables}) \times (\text{Loop Bound} + 1)$$

In our case, the loop bound is 2 and we have 2 nested loops. This problem can be extended, as shown in the example in Figure 3.5.

```
int main() {
    int a = 6, b = 6, c = 6, d = 6;
    for(a = 0; a < 6; ++a) {
        for(b = 0; b < 6; ++b) {
            for(c = 0; c < 6; ++c) {
                for(d = 0; d < 6; ++d) {

                }
            }
        }
    }
    assert(a == 6 && b == 6 && c == 6 && d == 6);
}
```

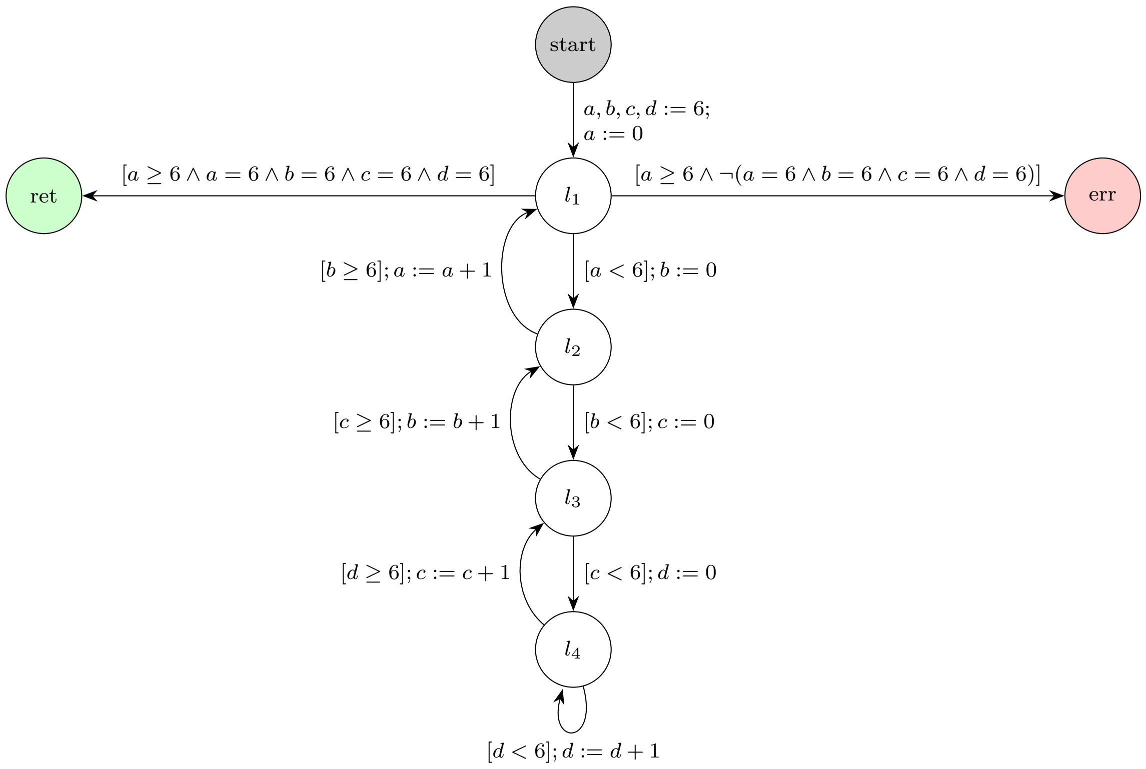
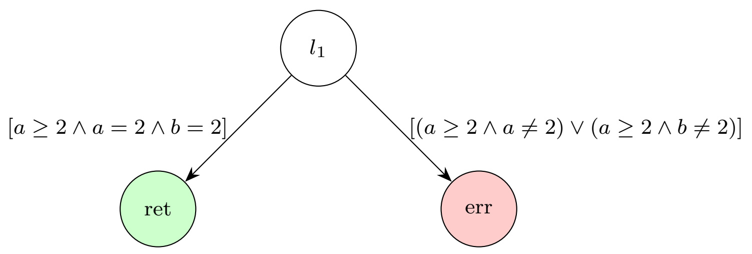


Figure 3.5: A CFA for the nested loop program leading to a final assertion state.

Here, the loop bound is 6 and the number of variables is 4. Therefore, the size of the precision would be approximately $|\Pi| = 4 \times (6 + 1) = 28$. The search space grows exponentially with the number of predicates (roughly $2^{|\Pi|}$), making the problem intractable for this approach.

To understand why K-Induction also fails, let's analyze a simplified version of the program.



l_1
$[a \geq 2 \land a = 2 \land b = 2]$
$[(a \geq 2 \land a \neq 2) \lor (a \geq 2 \land b \neq 2)]$
ret
err

Figure 3.6: The simplifed exit loop conditions for program 3.3.

The transition relation $T(s_k, s_{k+1})$ contains the logical expression leading to the error state as follows:

$$loc_k = l_1 \land ((a_k \geq 2 \land a_k \neq 2) \lor (a_k \geq 2 \land b_k \neq 2)) \land loc_{k+1} = \text{err}$$

Let's analyze the term $a_k \geq 2 \land a_k \neq 2$. The question in the inductive step is whether an initial state s_0 can be chosen such that this expression evaluates to true for some step k. K-Induction, without the program's initial state $I(s_0)$, is free to choose any value for a_0. If it chooses $a_0 \geq 3$, the loop is never entered, the exit condition $a \geq 2$ holds, but the assertion $a = 2$ fails, leading to a spurious counterexample.

Even more importantly, K-Induction cannot correlate the values of a and b. The term $a_k \geq 2 \land b_k \neq 2$ highlights this. In the inductive step, there is no constraint that forces b_k to be 2 just because a_k has reached 2. The solver can find a model where the first loop has completed ($a_k \geq 2$) but the second loop has not ($b_k < 2$), which again leads to a spurious counterexample. For K-Induction to verify this program, it has to unroll all the loops completely. If we have multiple nested loops, the number of required unrollings can be on the order of (loop bound)$^{(\text{number of loops})}$. For the example in Figure 3.5, this would be $6^4 = 1296$, which is a very large bound.

Now we will see how the invariants from the partial PredCart analysis can help K-Induction solve this problem. Let's suppose we let the CEGAR algorithm run for 3 iterations, which generates the ARG shown in Figure 3.4c. From this graph, we can extract the following location invariants:

$$\begin{aligned}\mathcal{I}_{\text{inv}} = \{&(\text{start}, \text{true}), (l_1, (a < 2) \lor \text{false} \lor (b \geq 2)),\\ &(l_2, (a < 2 \land b < 2) \lor (a < 2)), (l_3, \text{false} \lor (a \geq 2 \land b \geq 2)),\\ &(\text{final}, a \geq 2 \land b \geq 2), (\text{error}, a \geq 2 \land b \geq 2)\}\end{aligned}$$

Simplifying these expressions, we get:

$$\begin{aligned}\mathcal{I}_{\text{inv}} = \{&(\text{start}, \text{true}), (l_1, a < 2 \lor b \geq 2), (l_2, a < 2),\\ &(l_3, a \geq 2 \land b \geq 2), (\text{final}, a \geq 2 \land b \geq 2), (\text{error}, a \geq 2 \land b \geq 2)\}\end{aligned}$$

Now we can inject these invariants into the XCFA.

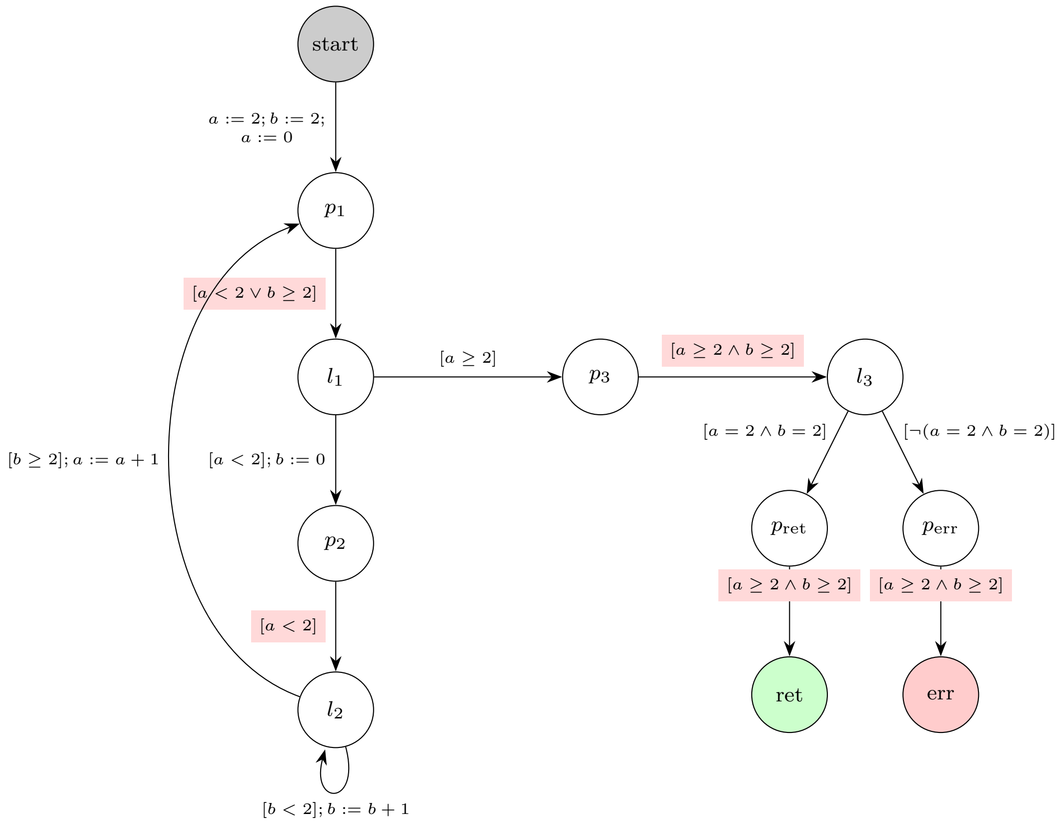


Figure 3.7: An XCFA with explicit intermediate locations (p_i) and injected invariant guards (highlighted in red).

We begin by analyzing why this problem is now easily solvable using k-induction. To facilitate a simpler analysis, we first simplify the exit loop condition.

For the path $l_1 \to p_3 \to l_3 \to p_{ret} \to ret$, the accumulated assumptions are:

$$(a \geq 2) \land (a \geq 2 \land b \geq 2) \land (a = 2 \land b = 2)$$

This expression can be simplified to:

$$a = 2 \land b = 2$$

Conversely, for the path $l_1 \to p_3 \to l_3 \to p_{err} \to err$, the accumulated assumptions are:

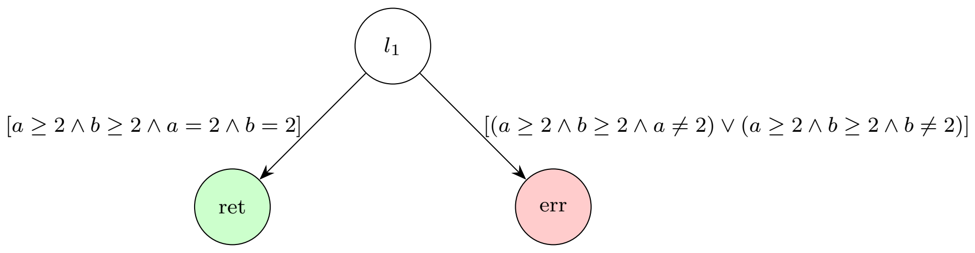
$$(a \geq 2) \land (a \geq 2 \land b \geq 2) \land \neg(a = 2 \land b = 2)$$

This can be simplified to:

$$(a \geq 2 \land b \geq 2 \land a \neq 2) \lor (a \geq 2 \land b \geq 2 \land b \neq 2)$$

Figure 3.8 shows the simplified loop assumptions in the injected XCFA.

Now let's analyze the guards on the path leading to the error location. The injected invariant $a \geq 2 \land b \geq 2$ at location l_3 is the key. It provides the *global property* that K-Induction was missing. K-Induction no longer needs to derive the fact that when the first loop terminates (i.e., $a \geq 2$), the second loop must also have terminated ($b \geq 2$). This result is injected directly into the XCFA.



l_1
$[a \geq 2 \land b \geq 2 \land a = 2 \land b = 2]$
$[(a \geq 2 \land b \geq 2 \land a \neq 2) \lor (a \geq 2 \land b \geq 2 \land b \neq 2)]$
ret
err

Figure 3.8: The simplifed exit loop conditions for program 3.7.

This can be seen in the disjunct $(a \geq 2 \land b \geq 2 \land b \neq 2)$. In the inductive step, K-Induction can still infer from its hypothesis that a and b are incremented by one in their respective loops. It can therefore deduce that upon exiting the loops, their values must be $a = 2$ and $b = 2$. When this local knowledge is combined with the injected invariant, the condition leading to the error state, which requires $a = 2$ and $b \neq 2$ (or vice versa), becomes a clear contradiction. The step case query becomes unsatisfiable, proving the safety of the program without requiring extensive loop unrolling.

3.5 Stopping Heuristics

Knowing when to stop an algorithm and transfer its partial results can significantly boost the performance of a portfolio. However, this is a difficult problem, as it requires predicting the future behavior of the algorithms.

In Example 3.4.1, we decided to run the algorithm for only two iterations, which was sufficient. However, for the more complex example in Figure 3.5, which has four nested loops with a larger loop bound of 6, running only two CEGAR iterations would be insufficient to extract useful invariants. Therefore, it is crucial to develop methods for predicting when an algorithm has run long enough to generate valuable partial results without wasting excessive time.

Presently, the most common tactic is to let an algorithm run for a predefined amount of time, essentially using a fixed timeout. Offline AI models have also been implemented to predict if an algorithm will exceed its time limit. In this work, we propose a lightweight, *online algorithm* to predict such a timeout specifically for predicate abstraction in CEGAR.

3.6 Using RLS Predictor to Predict Predicate Abstraction CEGAR Timeout

During the CEGAR execution, there are many dynamic parameters we can work with. Some of them include the size of the precision, the ARG size, the complexity of SMT queries (i.e., the time it takes to resolve an SMT query), refinement parameters, **iteration time**, and others.

First, let's analyze the size of the precision, $|\Pi|$. In predicate abstraction within CEGAR, $2^{|\Pi|}$ represents the number of possible states the program can be in with respect to the given precision. The problem with looking only at the precision size is that many of these states may be impossible to reach because the program never enters such configurations. Only a fraction of them are truly traversable. The exact number of traversable states cannot be known without exploring them, i.e., running the algorithm. Therefore, our heuristic will assume that the number of traversable states follows an exponential growth pattern with each iteration, a growth that is slower than the growth of all possible states, $2^{|\Pi|}$. An exponential growth is a reasonable estimate because the number of nodes in a path and the number of path in a program often expands exponentially .

Now, let's analyze the Abstract Reachability Graph (ARG) size. If we were to unroll the full ARG, we would explore all traversable states. However, CEGAR does not unroll the entire ARG; it only unrolls until the first error is found. The ARG size can give us information on how easily error states are found. A common issue with the ARG is that it can explode in size. This typically happens when all the easy to find error states have been discovered, by which time the precision has grown very large. The ARG must then prove safety with this large precision, which is the problem predicate abstraction faced in example B.2. The ARG size can thus be used to detect state space explosions. However, this has a limitation: there are cases where the state space explodes, but the SMT queries are so simple that the large state space is traversed very quickly.

We could analyze more parameters, but their interactions become very complex. The novel idea here is that all the aforementioned issues and parameters are implicitly contained and combined within the iteration time.

- Because the number of traversable states grows exponentially, the iteration time also shows an exponential tendency.
- It inherently includes the complexity of the SMT queries.
- It resolves the issue with the ARG, where a state explosion could be offset by easily traversable states, because of simple SMT queries.
- It contains information about the easiness of finding errors. The easier the error to find the smalle the iteration time, because the arg is small.
- It includes the refinement time.

The intuition is that all these parameters are somehow combined and expressed in the unit of time. Therefore, predicting a timeout based on the iteration time appears to be a very good and simple heuristic.

As mentioned, because the traversable state space grows exponentially, the iteration time also has an exponential tendency. Our proposed algorithm operates according to the following steps:

- A global upperbound timeout for the entire process is maintained.
- Upon the completion of each CEGAR iteration, a new data point is recorded, consisting of the iteration number and its measured execution time.
- A one dimensional RLS predictor, which uses the linearization technique for exponential relationships as described in Section 2.6.2, is updated with this new data point to predict the time required for the subsequent iteration.
- This predicted time is compared against the time remaining until the global timeout. If the prediction exceeds the available time, the algorithm is aborted; otherwise, the process repeats.

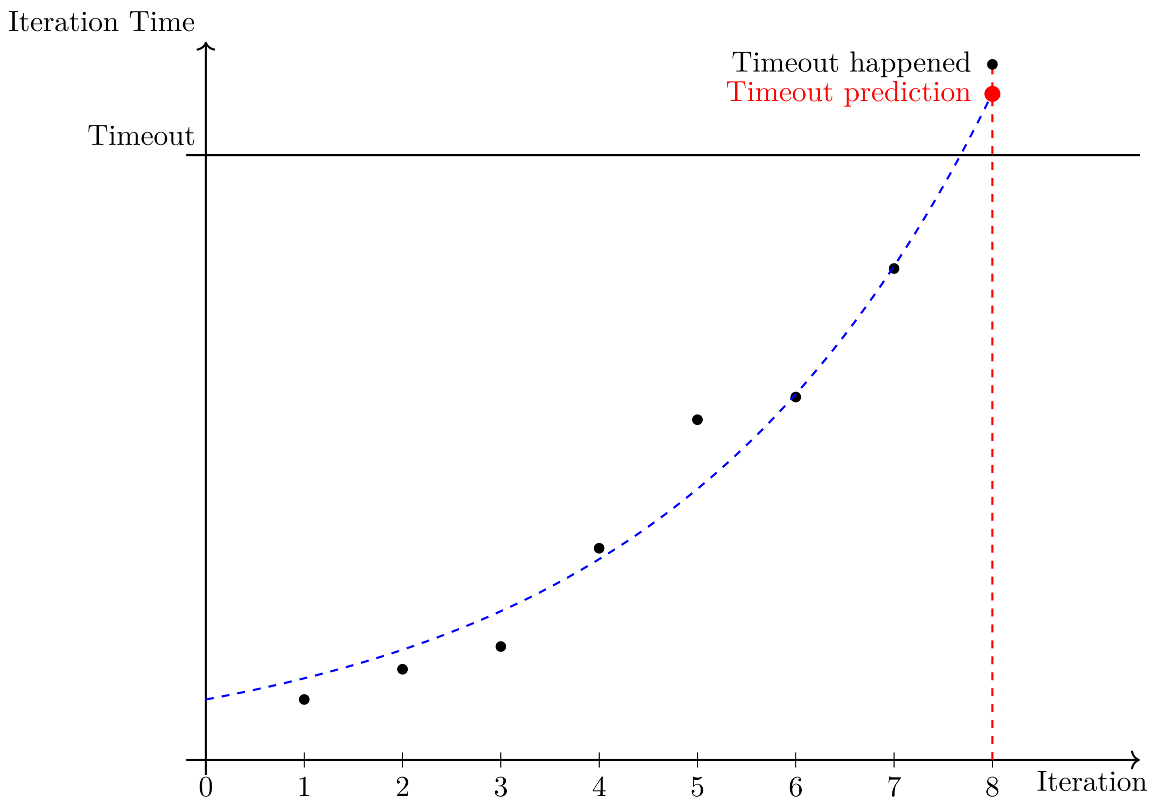


Figure 3.9: Illustration of the timeout prediction heuristic using RLS. The black dots represent the measured time for each completed CEGAR iteration. The RLS predictor fits an exponential curve (blue dashed line) to this data. After iteration 7, it predicts the time for the next iteration (red dot). As this prediction surpasses the global timeout threshold, the execution is preemptively aborted, successfully avoiding the actual timeout that would have occurred (upper black dot).

Predicting State Space Explosion

The question is, how do we predict a state space explosion? Currently, our implementation employs a simple method: if the actual time for a CEGAR iteration is significantly larger than the predicted time, we assume a state space explosion has occurred.

However, a more sophisticated approach would be to continuously monitor the time spent *within* a single iteration. Since our RLS calculations are not computationally expensive, we can update the timeout prediction multiple times during a single CEGAR iteration as its execution time grows. This could be implemented by first retracting the last data point added to the RLS model (i.e., replacing $w(t)$ with $w(t-1)$ and $P(t)$ with $P(t-1)$) and then adding a new data point with the currently elapsed time for the ongoing iteration. This check could be performed, for example, each time the ARG exploration completes the expansion of a node.

The main limitation of our current implementation is that updates occur only at the end of each iteration, meaning a state space explosion can be detected only *after* the costly iteration has finished. In contrast, the proposed *continuous update* approach enables earlier and more accurate detection during the iteration, potentially preventing a complete timeout of the algorithm.

Chapter 4

Evaluation

In this chapter, we will outline the design of the evaluation to test our newly proposed algorithms. We will then formulate our research questions and analyze the results in accordance with these questions. Finally, we will provide an overall discussion and address any threats to the validity of our findings.

4.1 Evaluation Design

To perform our evaluation, we will construct sequential portfolios of the following types:

1. $\text{KInd}(900)$
2. $\text{PredCart}(900)$
3. $\text{PredCart}(100) \xrightarrow{\text{any}} \text{KInd}()$
4. $\text{PredCart}(100, \text{pRes}) \xrightarrow{\text{any}} \text{KInd}()$
5. $\text{PredCart}(100, \text{pRes}, \text{Heu}) \xrightarrow{\text{any}} \text{KInd}()$
6. $\text{PredCart}(900, \text{pRes}) \xrightarrow{\text{any}} \text{KInd}()$
7. $\text{PredCart}(900, \text{pRes}, \text{Heu}) \xrightarrow{\text{any}} \text{KInd}()$

Where pRes is shorthand for partial results and Heu is shorthand for heuristic. As an example, the last portfolio is configured to run predicate abstraction for 900 seconds, passing the partial results while also utilizing the heuristic. If any error occurs, KInd is used as a fallback. For the timeout prediction heuristic, we will use the portfolios $\text{PredCart}(900, \text{pRes}) \xrightarrow{\text{any}} \text{KInd}()$ and $\text{PredCart}(900, \text{pRes}, \text{Heu}) \xrightarrow{\text{any}} \text{KInd}()$.

The heuristic's performance can be described using four categories: *True Positives (TP)* occur when it correctly predicts a timeout that indeed happens, while *True Negatives (TN)* indicate correctly predicting that no timeout occurs. In contrast, *False Positives (FP)* represent cases where a timeout is predicted but does not occur, and *False Negatives (FN)* capture situations where the heuristic fails to predict a timeout that actually happens.

Using these definitions, we will compute the following standard metrics:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FN + FP}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{Specificity} = \frac{TN}{TN + FP}$$

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Accuracy indicates the percentage of correct predictions made by our heuristic. However, this metric can be misleading if the number of actual timeouts is low, as is the case for our test suite.

Precision answers the question: when our heuristic predicts a timeout, how often is it correct? This helps gauge the reliability of our heuristic.

Recall addresses: of all the runs that were actually going to time out, what fraction did we successfully catch? This is useful for determining if our algorithm is effective at predicting timeouts when they occur.

Specificity tells us: of all the runs that were not going to time out, how many did our heuristic correctly identify as safe to continue?

Arguably, we are most concerned with how well our algorithm performs at predicting timeouts. Therefore, precision and recall are particularly important, as we want to catch as many timeouts as possible without being overly imprecise. To balance these two aspects, we use the **F1 score** as a single, combined metric.

4.1.1 Research Questions

Our evaluation aims to answer the following questions:

- **RQ1 (Effectiveness of the Partial Result Transfer Algorithm):** How many new programs, which could not be solved by either standalone algorithm, can our proposed partial result transfer algorithm solve?
- **RQ2 (Effectiveness of the Iteration Time Heuristic):** How effective is our proposed iteration time heuristic at predicting timeouts?

Benchmark Set

For our benchmark set, we utilize SV-Benchmarks, which contains a large number of C programs, in the version used by SV-COMP'25. From the reachability safety category, we selected the `ControlFlow`, `ECA`, and `Loops` subcategories, resulting in a total of 2774 test cases.

Benchmark Environment

To ensure the reliability of our evaluation, we employed `BENCHEXEC`, a reliable benchmarking framework, for our experiments. The experiments were conducted on virtual machines orchestrated by `BENCHCLOUD`. For the verification process, we adhered to the constraints of the SV-COMP (International Competition on Software Verification), imposing a time limit of 900 seconds, a memory limit of 15 GB, and restricting each run to 2 CPU cores.

Tool Support

The aforementioned algorithms and portfolios are implemented in THETA[1]. Within this framework, we implemented the partial result extraction from the Abstract Reachability Graph (ARG) in the CEGAR loop. The injection of partial results was implemented in the XCFA module of THETA. We also implemented in theta a one dimensional Recursive Least Squares (RLS) predictor with linearization for exponential properties, which is used to realize our iteration time heuristic algorithm. I store the results in github[2].

4.2 Analysis of Results

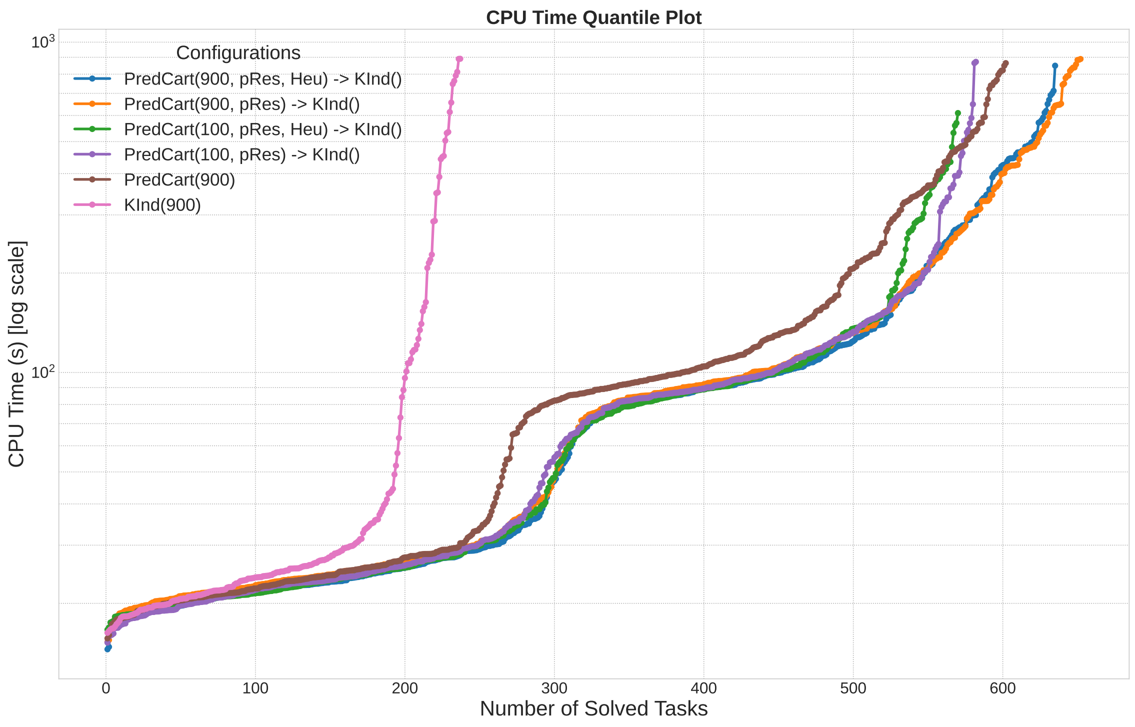


Figure 4.1: CPU time quantile plot comparing configurations of the proposed algorithms (PredCart and KInd) with and without partial result transfer and iteration time heuristic.

4.2.1 RQ1: Effectiveness of the Partial Result Transfer Algorithm

The graph in Figure 4.1 provides significant insights into our algorithm's performance. Let's first analyze the portfolios without the heuristic. The portfolio $\text{PredCart}(100, \text{pRes}) \xrightarrow{\text{any}} \text{KInd}()$ performs worse than the standalone PredCart(900) regarding the total number of test cases solved. On the other hand, it is faster for the test cases it does solve. This is mainly because K-Induction solves significantly fewer tasks than Predicate Abstraction with Cartesian Abstraction (PredCart). Consequently, allocating more time to PredCart, as in the $\text{PredCart}(900, \text{pRes}) \xrightarrow{\text{any}} \text{KInd}()$ portfolio, allows it to solve more tasks.

[1]`https://github.com/KlevisImeri/theta/tree/feature/partial-results`
[2]`https://github.com/KlevisImeri/thetaresults`

To answer our research question, we must evaluate how effective the partial results truly were. There are 2118 test cases where both PredCart(900) and KInd(900) fail. Out of these difficult instances, the PredCart(100, pRes) $\xrightarrow{\text{any}}$ KInd() portfolio solves ***13***. Furthermore, when comparing the portfolio versions with and without partial results, we see that out of the 1337 cases where PredCart(100) $\xrightarrow{\text{any}}$ KInd() timed out, the version with partial results, PredCart(100, pRes) $\xrightarrow{\text{any}}$ KInd(), was able to solve ***20*** of them.

These may not seem like large numbers, but the test cases where both PredCart and K-Induction fail are inherently hard problems. Additionally, combining different algorithms can introduce other implementation challenges.

On the other hand, the portfolio PredCart(900, pRes) $\xrightarrow{\text{any}}$ KInd() solves no new test cases. This is because partial results are only extracted when a timeout occurs. In this configuration, the timeout for the first stage happens only when the entire portfolio's time limit is reached, leaving no time for the second stage to run.

As seen in the graph, the overall performance difference between the portfolios with and without partial results is small, and the effects on the overall performance are negligible. However, the key advantage is the ability to solve test cases that were not solvable before in a reasonable amount of time. In conclusion, while our partial result transfer algorithm is not highly effective across the board, it successfully solves certain test cases that were previously unsolvable.

4.2.2 RQ2: Effectiveness of the Iteration Time Heuristic

To analyze the effectiveness of the iteration time heuristic, we will evaluate its performance metrics. We compare the actual timeouts observed in PredCart(900) with the predicted timeouts from the portfolio PredCart(900, pRes, Heu) $\xrightarrow{\text{any}}$ KInd(). This heuristic requires setting several initialization parameters, for which we used the following values: $w = 1.7$, $P(0) = 1000$, and $\lambda = 0.96$. The value $w = 1.7$ was chosen based on the intuition of how much the number of traversable states would increase, although in reality, we observed this factor to be closer to the 1.0 to 1.1 range. For the forgetting factor, λ, we chose a middle ground value to allow the algorithm to be more flexible when the behavior of the CEGAR loop changes abruptly. We set a high initial covariance of $P(0) = 1000$ because we have no prior information about the algorithm's behavior at the beginning and we want the algorithm to learn fast. After extracting the results, we found that 1737 cases failed for reasons other than a timeout. We, therefore, computed the metrics based on the remaining 1037 relevant cases.

Table 4.1: Heuristic Performance Metrics

Metric	**Value**
True Positives (TP)	290
True Negatives (TN)	575
False Positives (FP)	26
False Negatives (FN)	146
Accuracy	0.8341
Precision	0.9177
Recall	0.6651
Specificity	0.9567
F1 Score	0.7713

From these results, we observe that our algorithm achieves very high precision and specificity. This indicates that when the heuristic predicts a timeout, it is correct in most cases, and it is also very effective at identifying runs that will not time out.

On the other hand, the recall is somewhat lower. As mentioned in Subsection 3.6, this is mainly because we do not implement a continuous update of the heuristic, instead, it is only updated at the end of each iteration. Therefore, when a sudden state space explosion occurs, our heuristic can only predict a timeout after the costly iteration has already finished.

Even though there is significant room for improvement, our prediction algorithm performs quite well, achieving an F1 score of 0.7713.

4.3 Discussion of results

As seen from the evaluation, the algorithms we proposed show positive results. While the partial result transfer algorithm does not perform exceptionally well overall, it still solves 13 cases that were previously unsolvable. It is important to mention that implementation complexities can accumulate when algorithms are combined, making even a small number of newly solved cases a significant result.

On the other hand, the timeout prediction algorithm performs flawlessly, with very high precision and specificity. Although the recall is somewhat lower, this could potentially be improved with a more finegrained update of the heuristic during execution.

One hurdle in obtaining these results is the necessity of running many types of portfolios, which becomes computationally expensive, especially if mistakes are made in the setup. Another challenge is the presence of errors within THETA in certain scenarios. For example, as seen in Figure 4.1, the portfolios with timeout prediction do not perform better than those without it. The reason for this is not that the timeout prediction itself is flawed, but rather that there are no instances in our benchmark set where PredCart times out on a program that K-Induction can subsequently solve. This implies there was no opportunity for our heuristic to create a positive impact on the overall portfolio performance in these specific tests.

4.3.1 Threats to Validity

External Validity The generalizability of our conclusions is limited by the scope of the benchmarks evaluated. As the selected set of tasks was not exhaustive, the observed performance metrics may not be representative of the tools' behavior across all possible scenarios.

Internal Validity We established a robust and consistent measurement environment through the BenchExec framework, which leverages advanced Linux kernel features for reliable execution. Despite this, potential threats to precision remain. Uncontrolled environmental factors, such as temperature, and inherent nondeterminism in tool execution can lead to minor variations in CPU time. Furthermore, the risk of software bugs is amplified when algorithms are designed to cooperate.

Chapter 5

Related Work

Developing new model checking algorithms in formal verification is a very challenging task. Therefore, researchers often try to combine existing algorithms to exploit their individual advantages. This implies that the idea of combining algorithms and passing partial results between them has been extensively studied [3, 8, 12, 11]. The general approach for passing results is to use invariants, which can be combined to form a witness [2, 14, 12]. New witness formats are continually being refined within the community, and different methods for passing them within algorithms and between different verifiers are being explored.

Besides accepted witness formats, other forms have been proposed to pass partial results. For instance, CPAchecker [5] uses a so called assumption automaton [11] which, together with its framework of product CPAs, can be used to pass the already explored search space to a subsequent algorithm. This technique is known as Conditional Model Checking [11]. Both Conditional Model Checking with assumption automata and the exchange of witnesses have their own pros and cons. To address these issues, the General Intermediate Automaton was proposed [23].

Many verifiers do not natively support the injection of partial results, so Reducers [13, 15] have been proposed to overcome this limitation. Reducers inject the assumptions (invariants) directly into the source code, such that the subsequent verification tool does not need to traverse paths that have already been explored. Another cooperative method is dynamic program splitting [27], where a verifier, upon encountering a difficult problem, splits the task into subtasks and initiates verification on them in parallel or sequentially. Regarding the injection of invariants, work has also been done on integrating them into Interpolation Based Model Checking [17].

A significant amount of work has been done in the area of verifier cooperation, but these methods are often complex and difficult to implement. Sometimes, very general approaches are not necessary, and the goal is simply to pass invariants between different algorithms within a single verifier. The solution proposed in this work offers a simple yet effective way to achieve this.

To effectively pass partial results, it is crucial to determine how long an algorithm should run so that useful partial results can be extracted. This is a field with limited research. One study employed an offline AI model [28] to predict if the current algorithm will time out, and the model performed quite well. However, designing, training, and running such a model (including loading it into memory and the inference time) during execution is resource intensive. To combat this, we propose an online model that uses an RLS predictor [24] with linearization for the exponential relationships observed.

Chapter 6

Conclusion

6.0.1 Summary

In this work, we explored two main contributions to enhance sequential algorithm portfolios in software verification. First, we proposed a simple yet effective method for transferring partial results by extracting location invariants from an incomplete CEGAR run and injecting them as assumptions into the XCFA for a subsequent algorithm. Our evaluation demonstrated that while this technique did not yield a broad performance increase across all benchmarks, it was capable of solving a number of challenging verification tasks that were unsolvable by the standalone algorithms.

Second, we introduced a novel online heuristic, based on a one dimensional Recursive Least Squares predictor, to forecast timeouts in CEGAR. The heuristic achieved high precision and specificity, proving reliable in its predictions. However, its recall indicated room for improvement. This was primarily due to the heuristic being updated only at the end of each CEGAR iteration. Overall, our proposed heuristic and model reached an F1 score of approximately 77%.

6.0.2 Future Work

Building on the foundation of this research, several promising directions for future work have been identified:

- **Extensive Evaluation:** A more comprehensive evaluation on a wider range of benchmarks is necessary to better assess the generalizability of our findings and the robustness of the proposed techniques.
- **Enhanced Heuristics:** The timeout prediction heuristic could be significantly improved. This includes extending the one dimensional RLS predictor to a multi-dimensional model that incorporates more features, such as the complexity of the invariants being generated, to make more informed predictions.
- **Invariant Filtering:** Currently, all derived invariants are transferred. A crucial area for future research is the development of filtering mechanisms to select only the most beneficial invariants, such as loop invariants, to pass to the subsequent algorithm. This would reduce noise and focus the next stage's analysis.
- **Targeted Cooperation:** Instead of a global extraction, a more targeted approach could be developed. This would involve identifying regions of the code where the first

algorithm excels, extracting invariants specifically from those regions, and passing them to a second algorithm known to struggle with those particular code structures.

- **Deeper Algorithm Integration:** Our current method injects invariants at the XCFA level. A more powerful technique would be to inject these invariants directly into the internal workings of the subsequent algorithm, for example, by providing them as initial lemmas to K-Induction.
- **Inter-Tool Cooperation:** To broaden the applicability of our work, we plan to support standard exchange formats like Witness 2.0. This would enable passing partial results not just between algorithms within Theta, but also to entirely different verification frameworks, leveraging the strengths of the wider verification ecosystem.

Bibliography

[1] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. *An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment*, page 254–268. Springer Berlin Heidelberg, 2005. ISBN 9783540320302. DOI: 10.1007/11560548_20. URL http://dx.doi.org/10.1007/11560548_20.

[2] Paulína Ayaziová, Dirk Beyer, Marian Lingsch-Rosenfeld, Martin Spiessl, and Jan Strejček. *Software Verification Witnesses 2.0*, page 184–203. Springer Nature Switzerland, October 2024. ISBN 9783031661495. DOI: 10.1007/978-3-031-66149-5_11. URL http://dx.doi.org/10.1007/978-3-031-66149-5_11.

[3] Dirk Beyer. *Partial Verification and Intermediate Results as a Solution to Combine Automatic and Interactive Verification Techniques*, page 874–880. Springer International Publishing, 2016. ISBN 9783319471662. DOI: 10.1007/978-3-319-47166-2_60. URL http://dx.doi.org/10.1007/978-3-319-47166-2_60.

[4] Dirk Beyer and Sudeep Kanav. *CoVeriTeam: On-Demand Composition of Cooperative Verification Systems*, page 561–579. Springer International Publishing, 2022. ISBN 9783030995249. DOI: 10.1007/978-3-030-99524-9_31. URL http://dx.doi.org/10.1007/978-3-030-99524-9_31.

[5] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

[6] Dirk Beyer and Jan Strejček. *Improvements in Software Verification and Witness Validation: SV-COMP 2025*, page 151–186. Springer Nature Switzerland, 2025. ISBN 9783031906602. DOI: 10.1007/978-3-031-90660-2_9. URL http://dx.doi.org/10.1007/978-3-031-90660-2_9.

[7] Dirk Beyer and Heike Wehrheim. Verification artifacts in cooperative verification: Survey and unifying component framework. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part I*, page 143–167, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-61361-7. DOI: 10.1007/978-3-030-61362-4_8. URL https://doi.org/10.1007/978-3-030-61362-4_8.

[8] Dirk Beyer and Philipp Wendler. *Reuse of Verification Results: Conditional Model Checking, Precision Reuse, and Verification Witnesses*, page 1–17. Springer Berlin Heidelberg, 2013. ISBN 9783642391767. DOI: 10.1007/978-3-642-39176-7_1. URL http://dx.doi.org/10.1007/978-3-642-39176-7_1.

[9] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*, page 504–518. Springer Berlin Heidelberg. ISBN 9783540733676. DOI: 10.1007/978-3-540-73368-3_51. URL http://dx.doi.org/10.1007/978-3-540-73368-3_51.

[10] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5–6):505–525, September 2007. ISSN 1433-2787. DOI: 10.1007/s10009-007-0044-z. URL http://dx.doi.org/10.1007/s10009-007-0044-z.

[11] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: a technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316149. DOI: 10.1145/2393596.2393664. URL https://doi.org/10.1145/2393596.2393664.

[12] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 326–337, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. DOI: 10.1145/2950290.2950351. URL https://doi.org/10.1145/2950290.2950351.

[13] Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. Reducer-based construction of conditional verifiers. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1182–1193, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. DOI: 10.1145/3180155.3180259. URL https://doi.org/10.1145/3180155.3180259.

[14] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4), September 2022. ISSN 1049-331X. DOI: 10.1145/3477579. URL https://doi.org/10.1145/3477579.

[15] Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. Decomposing software verification into off-the-shelf components: an application to cegar. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 536–548, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. DOI: 10.1145/3510003.3510064. URL https://doi.org/10.1145/3510003.3510064.

[16] Dirk Beyer, Nian-Ze Lee, and Philipp Wendler. Interpolation and sat-based model checking revisited: Adoption to software verification, 2022. URL https://arxiv.org/abs/2208.05046.

[17] Dirk Beyer, Po-Chun Chien, and Nian-Ze Lee. Augmenting interpolation-based model checking with auxiliary invariants. In Thomas Neele and Anton Wijs, editors, *Model Checking Software*, pages 227–247, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-66149-5.

[18] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic Model Checking without BDDs*, page 193–207. Springer Berlin Heidelberg, 1999. ISBN 9783540490593. DOI: 10.1007/3-540-49059-0_14. URL http://dx.doi.org/10.1007/3-540-49059-0_14.

[19] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003. ISSN 0004-5411. DOI: 10.1145/876638.876643. URL https://doi.org/10.1145/876638.876643.

[20] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer International Publishing, 2018. ISBN 9783319105758. DOI: 10.1007/978-3-319-10575-8. URL http://dx.doi.org/10.1007/978-3-319-10575-8.

[21] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. *Software Verification Using k-Induction*, page 351–368. Springer Berlin Heidelberg, 2011. ISBN 9783642237027. DOI: 10.1007/978-3-642-23702-7_26. URL http://dx.doi.org/10.1007/978-3-642-23702-7_26.

[22] Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, November 2019. ISSN 1573-0670. DOI: 10.1007/s10817-019-09535-x. URL http://dx.doi.org/10.1007/s10817-019-09535-x.

[23] Jan Haltermann and Heike Wehrheim. Exchanging information in cooperative software validation. *Software and Systems Modeling*, 23(3):695–719, Jun 2024. ISSN 1619-1374. DOI: 10.1007/s10270-024-01155-3. URL https://doi.org/10.1007/s10270-024-01155-3.

[24] Simon Haykin. *Adaptive Filter Theory*. Pearson Education, Upper Saddle River, NJ, 5th edition, 2014. ISBN 978-0132679583.

[25] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):1–54, October 2009. ISSN 1557-7341. DOI: 10.1145/1592434.1592438. URL http://dx.doi.org/10.1145/1592434.1592438.

[26] K. L. McMillan. *Interpolation and SAT-Based Model Checking*, page 1–13. Springer Berlin Heidelberg, 2003. ISBN 9783540450696. DOI: 10.1007/978-3-540-45069-6_1. URL http://dx.doi.org/10.1007/978-3-540-45069-6_1.

[27] Cedric Richter, Marek Chalupa, Marie-Christine Jakobs, and Heike Wehrheim. Cooperative software verification via dynamic program splitting. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2087–2099, 2025. DOI: 10.1109/ICSE55347.2025.00092.

[28] Nicola Thoben, Jan Haltermann, and Heike Wehrheim. *Timeout Prediction for Software Analyses*, page 340–358. Springer Nature Switzerland, 2023. ISBN 9783031471155. DOI: 10.1007/978-3-031-47115-5_19. URL http://dx.doi.org/10.1007/978-3-031-47115-5_19.

[29] Ákos Hajdu. *Effective Domain-Specific Formal Verification Techniques*. PhD thesis, Budapest University of Technology and Economics, Budapest, June 2020. URL http://hdl.handle.net/10890/13523. Ph.D. Dissertation, supervised by Zoltán Micskei.

Appendix A

Derivation of the RLS Covariance Update

A.1 Using the Sherman Morrison Formula to Calculate P(t)

To derive the recursive update rule for the covariance $P(t)$, we start from its definition in relation to $P(t-1)$:

$$P(t) = \left(\lambda R(t-1) + x_t^2\right)^{-1} = \left(\lambda P(t-1)^{-1} + x_t^2\right)^{-1}$$

This expression has the form $(A+uv^T)^{-1}$, which allows us to apply the Sherman Morrison formula.

The Sherman Morrison formula states:

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1} u}$$

For our specific, scalar case, we identify the terms as:

- $A = \lambda P(t-1)^{-1}$
- $u = x_t$
- $v^T = x_t$

First, we find the inverse of A:

$$A^{-1} = (\lambda P(t-1)^{-1})^{-1} = \frac{1}{\lambda}(P(t-1)^{-1})^{-1} = \frac{1}{\lambda} P(t-1)$$

Now, we substitute these components into the Sherman Morrison formula to derive the update rule for $P(t)$.

$$
\begin{aligned}
P(t) &= A^{-1} - \frac{A^{-1}uu^T A^{-1}}{1 + u^T A^{-1}u} \\
&= \frac{1}{\lambda}P(t-1) - \frac{(\frac{1}{\lambda}P(t-1))x_t x_t(\frac{1}{\lambda}P(t-1))}{1 + x_t(\frac{1}{\lambda}P(t-1))x_t} \\
&= \frac{1}{\lambda}P(t-1) - \frac{\frac{1}{\lambda^2}P(t-1)^2 x_t^2}{1 + \frac{1}{\lambda}P(t-1)x_t^2}
\end{aligned}
$$

Multiply the numerator and denominator of the fraction by λ:

$$
= \frac{1}{\lambda}P(t-1) - \frac{\frac{1}{\lambda}P(t-1)^2 x_t^2}{\lambda + P(t-1)x_t^2}
$$

Combine terms using a common denominator $\lambda(\lambda + P(t-1)x_t^2)$:

$$
\begin{aligned}
&= \frac{P(t-1)(\lambda + P(t-1)x_t^2) - P(t-1)^2 x_t^2}{\lambda(\lambda + P(t-1)x_t^2)} \\
&= \frac{\lambda P(t-1) + P(t-1)^2 x_t^2 - P(t-1)^2 x_t^2}{\lambda(\lambda + P(t-1)x_t^2)} \\
&= \frac{\lambda P(t-1)}{\lambda(\lambda + P(t-1)x_t^2)}
\end{aligned}
$$

This gives the final recursive update formula for $P(t)$:

$$
P(t) = \frac{P(t-1)}{\lambda + P(t-1)x_t^2}
$$

Appendix B

Supplementary Examples

B.1 Example Where Predicate Abstraction Verifies the Program and K-Induction Does Not

Figure B.1 presents an example program that is easily verified by CEGAR with predicate abstraction, but proves difficult for *k*-induction.

```
int main() {
  int n=nondet_uchar(), sn=0, i=1;
  while (i<=n) {
    sn = sn + 2;
    i++;
  }
  assert(sn==n*2 || sn == 0);
}
```

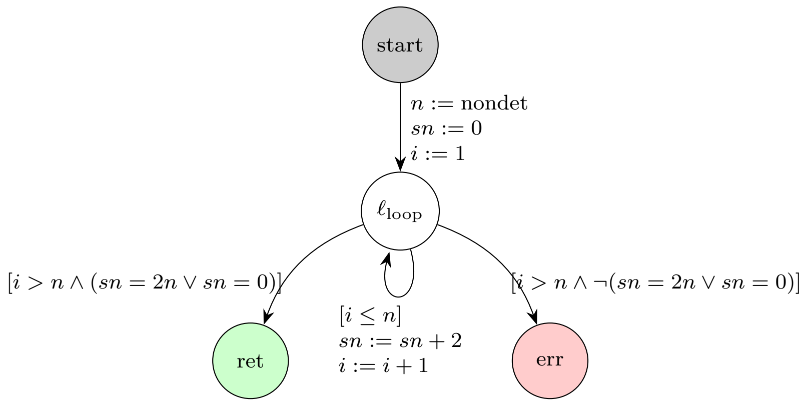


Figure B.1: A program and its CFA which CEGAR predicate abstraction can solve easily, while it is hard for (k)-induction.

In this program, if the input n is positive, the loop calculates the sum by adding 2 exactly n times, resulting in $sn = 2n$. If n is not positive, sn remains 0. To prove the assertion correct, a verifier must discover the loop invariant $sn = 2(i - 1)$ and recognize that upon loop exit, $i = n + 1$. Combining these facts confirms the post condition $sn = 2n$.

For CEGAR with predicate abstraction, this task is straightforward. The *refinement* step would quickly discover the necessary predicates, such as $sn = 2(i-1)$, to form the required precision. The abstraction mechanism is well suited for identifying and eliminating the infeasible paths corresponding to spurious counterexamples.

Conversely, this problem is challenging for *k-induction*. The structure of the program is similar to the one shown in Figure 2.3. Consequently, the structure of the inductive step query is also very similar, with the primary difference being the assumption on the loop's transition condition. As with the previous example, the *local* nature of the inductive step prevents it from using the initial conditions of the variables to prove the property.

$$\mathbf{\Psi}_k = \underbrace{(\mathrm{loc}_1 \neq \ell_{\mathrm{err}})}_{\mathbf{P}(s_1)} \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{T}(s_{k-1}, s_k) \wedge \underbrace{(\mathrm{loc}_k \neq \ell_{\mathrm{err}})}_{\mathbf{P}(s_k)}$$

$$\wedge \underbrace{\begin{pmatrix} \mathrm{loc}_k = \ell_{\mathrm{loop}} \wedge i_k \leq n_k \wedge \mathrm{loc}_{k+1} := \ell_{\mathrm{loop}} \wedge sn_{k+1} := sn_k + 2 \wedge i_{k+1} := i_k + 1 \\ \vee\ \mathrm{loc}_k = \ell_{\mathrm{loop}} \wedge i_k > n_k \wedge (sn_k = 2n_k \vee sn_k = 0) \wedge \mathrm{loc}_{k+1} := \ell_{\mathrm{ret}} \\ \vee\ \mathrm{loc}_k = \ell_{\mathrm{loop}} \wedge i_k > n_k \wedge \neg(sn_k = 2n_k \vee sn_k = 0) \wedge \mathrm{loc}_{k+1} := \ell_{\mathrm{err}} \end{pmatrix}}_{\mathbf{T}(s_k, s_{k+1})}$$

$$\wedge \underbrace{(\mathrm{loc}_{k+1} = \ell_{\mathrm{err}})}_{\overline{\mathbf{P}(s_{k+1})}}$$

The condition for reaching the error state would be $i_{k+1} > n \wedge sn_{k+1} \neq 2n \wedge sn_{k+1} \neq 0$, where k is the bound on the unrolled path length.

When examining the inductive step query for a given bound k (e.g., $k \leq 255$, the maximum value of an unsigned char), the SMT solver can easily find a model that satisfies the loop exit condition, $i_k > n$. This is readily achieved because, in the inductive step, the state at the beginning of the path is not constrained by the program's initial state. The solver is free to choose, for instance, a starting state for the sequence where $i_1 = 1$ and $n = k - 1$.

The second part of the condition is $sn_{k+1} \neq 2n$. By unrolling the loop k times, the SMT solver can establish the relation $sn_{k+1} = sn_k + 2 = \cdots = sn_1 + 2k$, where sn_1 is the value of sn at the beginning of the k-step sequence. However, sn_1 is treated as a *free variable* in the inductive step query. This is a fundamental aspect of k-induction, which is designed to prove that a property is an invariant for *all* paths of length k, regardless of the initial state. Since the program's initial state (where sn is initialized to 0) is not part of the query, the transition relation alone does not guarantee the required invariant $sn = 2n$. The solver can therefore find a counterexample, causing the inductive step to fail.

B.2 Example Where K-Induction Verifies the Program and Predicate Abstraction Does Not

It is possible that CEGAR predicate abstraction fails while k-induction succeeds in verifying a program. We process to showing such an examle.

The program above emulates acquiring and releasing many locks in parallel. It first enters a phase where it conditionally acquires the locks and then a second phase where it releases them. The nondeterministic value p_i determines whether the lock lk_i is acquired. Predicate abstraction will find abstract counterexamples to the error assertions. Along these paths, it will traverse the branches where a lock is taken (e.g., $p_i \neq 0$ and lk_i is set to 1). Since these are not real errors, the refiner will identify the counterexamples as *spurious*. Consequently, the precision will be updated to include predicates such as $p_i = 0$ and $lk_i = 1$. After the analysis has found all such spurious counterexamples, the precision set might look like this:

$$\Pi = \{lk_1 = 1, p_1 = 0, lk_2 = 1, p_2 = 0, \ldots, lk_n = 1, p_n = 0\}$$

With this refined precision, the abstraction must explore the state space to prove safety. This becomes a problem for predicate abstraction because each p_i is nondeterministic. The analysis must consider all combinations of the predicates for each lock. This implies checking 2^n permutations for n locks, resulting in an exponential complexity and a *state space explosion*.

On the other hand, *K-Induction* excels at verifying this program. This is because all locks are *independent* of one another. As a result, the clauses generated in the SMT induction step query are also independent, a structure that is easily exploited by modern *SMT solvers*. Below, the SMT induction step query is shown, with an emphasis on the state within a release node of a lock.

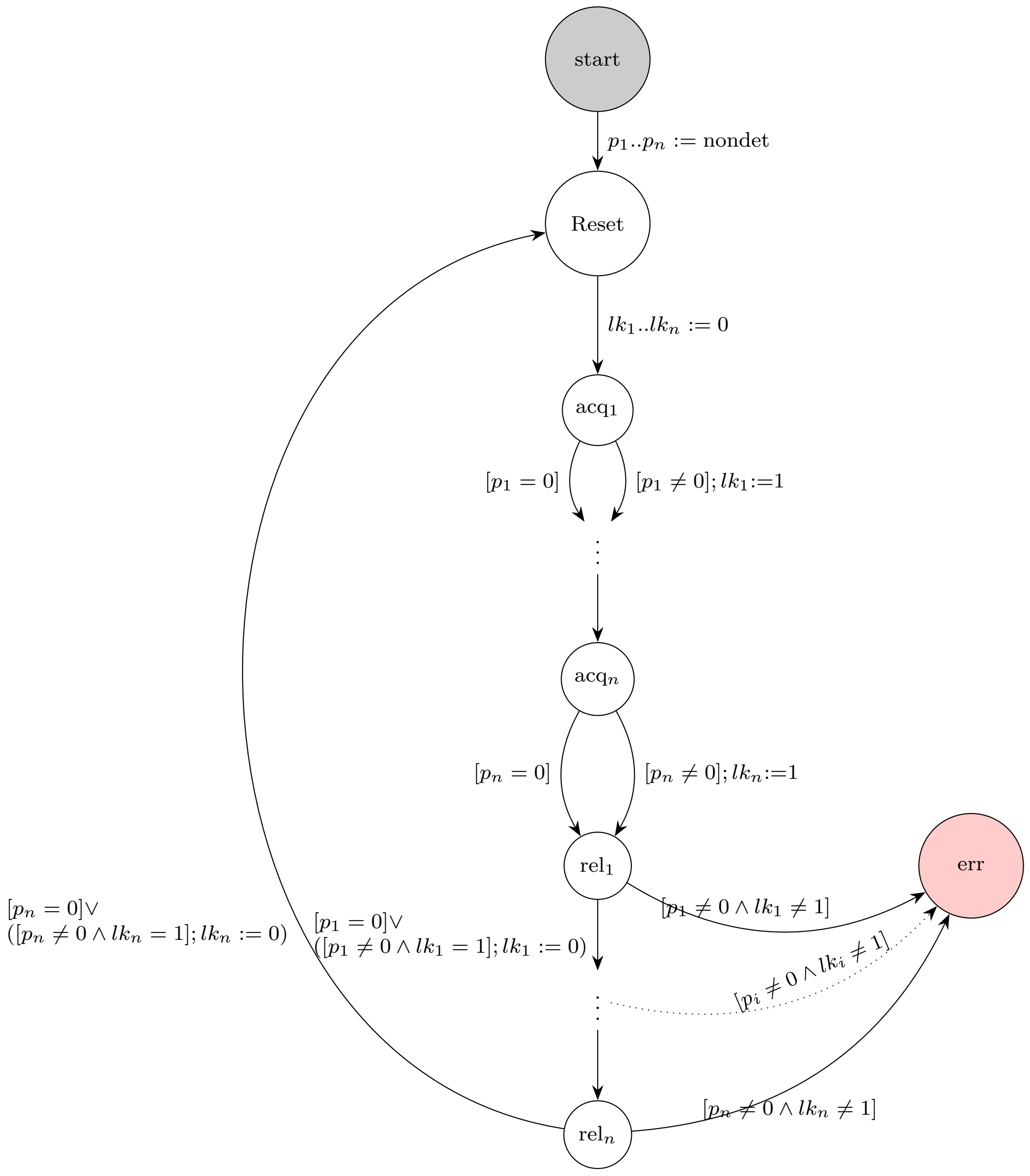
$$\mathbf{\Psi}_k = \underbrace{(\mathrm{loc}_1 \neq \ell_{\mathrm{err}})}_{\mathbf{P}(s_1)} \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{T}(s_{k-1}, s_k) \wedge \underbrace{(\mathrm{loc}_k \neq \ell_{\mathrm{err}})}_{\mathbf{P}(s_k)}$$

$$\wedge \underbrace{\begin{pmatrix} \ldots \\ \vee\ (\mathrm{loc}_k = \ell_{\mathrm{rel},j} \wedge p_{j,k} = 0 \wedge \mathrm{loc}_{k+1} := \ell_{\mathrm{rel},j+1}) \\ \vee\ (\mathrm{loc}_k = \ell_{\mathrm{rel},j} \wedge p_{j,k} \neq 0 \wedge lk_{j,k} = 1 \wedge \mathrm{loc}_{k+1} := \ell_{\mathrm{rel},j+1}) \\ \vee\ \underbrace{(\mathrm{loc}_k = \ell_{\mathrm{rel},j} \wedge p_{j,k} \neq 0 \wedge lk_{j,k} \neq 1 \wedge \mathrm{loc}_{k+1} := \ell_{\mathrm{err}})}_{\text{evaluates to false}} \\ \ldots \end{pmatrix}}_{\mathbf{T}(s_k, s_{k+1})}$$

$$\wedge \underbrace{(\mathrm{loc}_{k+1} = \ell_{\mathrm{err}})}_{\overline{\mathbf{P}(s_{k+1})}}$$

The SMT solver will be unable to find a satisfying assignment for the query. This is because for any arbitrary path of length k, the program's logic encoded in the transition relations implies that if execution reaches the release point for lock j where $p_j \neq 0$, then the state must be such that $lk_j = 1$. This directly contradicts the condition required to trigger the error, namely $lk_j \neq 1$. Consequently, the part of the formula modeling a transition to an error state becomes unsatisfiable.

```
int main() {
  int p1=nondet_int(), lk1; ...; int pn=nondet_int(), lkn;
  while(1) {
    // reset locks
    lk1=0; ...; lkn=0;
    // lock phase
    if (p1!=0) {lk1=1;} ...; if (pn!=0) {lkn=1;}
    // unlock phase
    if (p1!=0) {if(lk1!=1) ERROR; lk1=0;}
    ...
    if (pn!=0) {if(lkn!=1) ERROR; lkn=0;}
  }
}
```



start
$p_1..p_n :=$ nondet
Reset
$lk_1..lk_n := 0$
acq$_1$
$[p_1 = 0]$
$[p_1 \neq 0]; lk_1{:=}1$
acq$_n$
$[p_n = 0]$
$[p_n \neq 0]; lk_n{:=}1$
rel$_1$
err
$[p_1 \neq 0 \wedge lk_1 \neq 1]$
$[p_n = 0]\vee$
$([p_n \neq 0 \wedge lk_n = 1]; lk_n := 0)$
$[p_1 = 0]\vee$
$([p_1 \neq 0 \wedge lk_1 = 1]; lk_1 := 0)$
$[p_i \neq 0 \wedge lk_i \neq 1]$
$[p_n \neq 0 \wedge lk_n \neq 1]$
rel$_n$

Figure B.2: A detailed CFA for the locking protocol with explicit branching and a single error state.