Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Product abstraction-based strategies for efficient software model checking

Scientific Students' Association Report

Author:

Viktória Dorina Bajkai

Advisor:

Ákos Hajdu
BME MIT
MTA-BME Lendület Cyber-Physical Systems Research Group

2018

Contents

Kivonat **2**

Abstract **3**

1 Introduction **4**

2 Background **6**

2.1 Control Flow Automata . . . 6

2.2 Counterexample-guided abstract refinement . . . 8

2.2.1 Abstraction . . . 9

2.2.2 Refinement . . . 15

3 Product abstraction strategies **16**

3.1 Limit number of successors based on a single state . . . 17

3.2 Limit number of values on a path . . . 18

3.3 Limit number of values in ARG . . . 20

3.4 Related work . . . 20

4 Evaluation **22**

4.1 Implementation . . . 22

4.2 Measurement configuration . . . 23

4.3 Evaluate different k values for each strategy . . . 23

4.3.1 Single state-based strategy . . . 23

4.3.2 Path-based strategy . . . 25

4.3.3 ARG-based strategy . . . 26

4.4 Compare the best strategies . . . 28

4.5 Compare all strategies with each other . . . 29

5 Conclusion **31**

Bibliography **33**

Kivonat

Mindennapi életünket egyre jobban meghatározzák a szoftverrendszerek. Ezek sokszor biztonságkritikusak (pl. autonóm járművek, erőművek), tehát helyes működésük garantálása kiemelten fontos feladat. Ennek egyik eszköze a formális verifikáció, ami a hibák jelenlétét és a helyes működést is képes matematikailag precíz módon bizonyítani. Az egyik legelterjedtebb formális verifikációs módszer a modellellenőrzés, amely a program összes lehetséges állapotát és átmenetét (azaz állapotterét) szisztematikusan megvizsgálja. A módszer egyik hátránya viszont a nagy számítási igénye, ami gyakran megakadályozza használatát valós szoftvereken.

Az ellenpélda-alapú absztrakciófinomítás (angolul Counterexample-Guided Abstraction Refinement, CEGAR) egy olyan kiegészítő technika, melynek segítségével a modellellenőrzés hatékonyabbá tehető. Működése során a CEGAR iteratívan hozza létre és finomítja az ellenőrzendő probléma egy absztrakcióját. Az irodalomban több különböző absztrakciós megközelítés létezik, például az explicit változók módszere, illetve a predikátumabsztrakció. Előbbi a programnak csak a verifikáció céljából releváns változóit tartja nyilván, míg az utóbbi konkrét értékek helyett matematikai kifejezések teljesülését vizsgálja. Korábbi eredmények alapján megfigyelhető, hogy különböző absztrakciós módszerek különböző típusú szoftvereken működnek hatékonyabban. Ebből kifolyólag létrejöttek úgynevezett szorzat absztrakciók, amik többféle módszert kombinálnak egy algoritmusban.

Munkám során eltérő stratégiák alapján kombináltuk az explicit változókat predikátumokkal. Megközelítésünk lényege, hogy a már felderített absztrakt állapottérből kinyert információk figyelembe vételével a további felderítést és ellenőrzést hatékonyabbá teszi. Ezeket az új stratégiákat a Theta nevű nyílt forráskódú verifikációs keretrendszerben implementáltuk. Ennek segítségével szoftverrendszerek széles skáláján tudtuk lefuttatni méréseinket, többek között ipari vezérlő (PLC) kódokon. Összevetettük a különböző stratégiák előnyeit és hátrányait, és a már létező módszerekkel is összehasonlítottuk őket. Az eredményeink azt mutatják, hogy az új módszereink hatékonyan tudják kombinálni a meglévő algoritmusok előnyeit.

Abstract

Software systems are controlling devices that surround us in our everyday life. Many of these systems are safety-critical (e.g., autonomous vehicles, power plants), thus ensuring their correct operation is gaining increasing importance. Formal verification techniques can both reveal errors and give guarantees on correctness with a sound mathematical basis. One of the most widely used formal verification approaches is model checking, which systematically examines all possible states and transitions (i.e., the state space) of the software. However, a major drawback of model checking is its high computational complexity, often preventing its application on real-life software.

Counterexample-guided abstraction refinement (CEGAR) is a supplementary technique, making model checking more efficient in practice. CEGAR works by iteratively constructing and refining abstractions in a given abstract domain. There are several existing domains, such as explicit-values, which only track a relevant subset of program variables and predicates, which use logical formulas instead of concrete values. Observations show that different abstract domains are more suitable for different kinds of software systems. Therefore, so-called product domains have also emerged that combine different domains into a single algorithm.

In this work, we develop and examine various strategies to combine the explicit-value domain with predicates. Our approaches use different information from the already explored abstract state space to guide further exploration more efficiently. We implement our new strategies on top of Theta, an open source verification framework. This allows us to perform an experiment with a wide range of software systems including industrial PLC codes. We evaluate the strengths and weaknesses of the different approaches and we also compare them to existing methods. Our experiment shows that the new strategies can form efficient combinations of the existing algorithms.

Chapter 1

Introduction

Nowadays our reliance on safety-critical software systems is rapidly increasing. Therefore, there is a growing need for reliable proofs of their correct behaviour, since a failure can lead to serious damage. A promising approach for giving such proofs is formal software verification. Formal verification provides a sound mathematical basis to prove the correct operation of the programs with mathematical precision. A widely used formal verification method is *model checking*, which analyses the possible states and transitions (i.e., the state space) of the software for every possible input and checks whether certain properties are satisfied. There is a wide variety of properties that can be examined, including the failure of assertions, overflow and null pointers. The advantage of model checking is that it can not only reveal faults, but prove their absence as well. However, a major drawback is that systematically examining every possible state and transition for each input is too expensive computationally. Even for relatively simple programs the state space can be large or even infinite, which is called the "state space explosion". Various techniques have been developed in the past decades to overcome this problem. In our work, we use the supplementary technique *counterexample-guided abstraction refinement* (CEGAR).

CEGAR is a widely used software model checking algorithm, which uses abstraction to represent the state space in a more compact way. Abstraction means hiding certain details about the program. However this does not only yield a smaller state space, but we also lose information about the program. The abstraction usually over-approximates the original program. This means, that if no erroneous behaviour (i.e., counterexample) can be found in the abstraction, then the original program is also safe. However, losing information can also lead to finding a counterexample in the abstraction, that does not exist originally. That means, that the abstraction has to be refined to exclude the spurious counterexample. CEGAR usually starts with a coarse initial abstraction of the program and automatically finds the proper abstraction by a series of refinement steps.

CEGAR can work with different abstraction methods, such as explicit-value analysis and predicate abstraction. Explicit-value analysis operates by tracking values of only a subset of the program's variables, while predicate abstraction focuses on tracking certain facts (predicates) about the variables. However, different abstraction methods are more suitable for different kinds of software. Combinations of abstract domains, called product abstractions can unify the strengths of the different approaches. However, a key challenge is to find the proper way of combining them.

In our work, we develop a product abstraction algorithm, which combines explicit-value analysis and predicate abstraction. We try to focus on the advantages of both algorithms to propose three different strategies. These approaches use different information from the abstract state space (e.g., single states, paths, or all states) to combine explicit-value analysis with predicate abstraction efficiently.

In order to evaluate and compare these strategies, we implement them in THETA, an open source verification framework. We evaluate the performance of the new algorithms on multiple types of programs, including industrial programmable logic controller (PLC) codes from CERN, and several types of programs from the Competition on Software Verification (SV-Comp). We also compare the new strategies with the existing explicit-value analysis and predicate abstraction methods. The results show that our new algorithms can combine the advantages and outperform the existing methods.

Chapter 2

Background

In this chapter we present the background of product abstraction-based software model checking. We describe programs using Control Flow Automata (Section 2.1), a formal representation based on graphs and first order logic formulas. Then we introduce abstraction and the CEGAR approach (Section 2.2), which is a widely used technique for software verification.

2.1 Control Flow Automata

Programs can be described in various ways. Humans usually work with source code as it is readable and understandable. Computers on the other hand mostly work with a compiled binary, which can be executed. For verification purposes some formal representation is required, which allows mathematical reasoning. A widely used formal representation is the Control Flow Automata (CFA). It is also called a *model* of the program. A CFA is a graph-based representation annotated with first order logic (FOL) [12] formulas to describe the operations of the program. Given a domain D, let FOL^D denote formulas of that domain. For example, FOL^B denotes Boolean formulas like $x = y \wedge y > 5$.

While FOL is undecidable in general [12], in practice *satisfiability modulo theory* (SMT) solvers [2, 18] can efficiently reason about FOL formulas in some theories (e.g., integer arithmetic, arrays). In our work, we also use SMT solvers to reason about formulas.

Definition 1 (Control Flow Automata). A control flow automata [7] is a tuple $CFA = (V, L, l_0, E)$ where

- $V = \{v_1, v_2, \dots, v_n\}$ is a set of program variables with domains $D_1, D_2, \dots, D_n$,
- $L = \{l_1, l_2, \dots, l_k\}$ is a set of program locations representing the program counter,
- $l_0 \in L$ is the initial location, i.e., the entry point of the program,
- $E \subseteq L \times Ops \times L$ is a set of directed edges between locations, representing the operations, which are executed when we go from the source location to the target in the program. ▪

The operations $op \in Ops$ can be *assumptions, assignments or havocs.* Assumptions are boolean expressions (also called predicates) denoted by $[\varphi]$ where $\varphi \in FOL^B$. If there is an edge between two locations with an assumption, the program can take a transition to the target location if the predicate holds in the source location.

Assignments are in the form $v_i := \psi$, where $v_i \in V$ and $\psi \in FOL^{D_i}$. After this operation, v_i will be assigned the result of evaluating ψ in the target location. All other variables will have the same value as in the source location.

Havocs have the form of *havoc* v_i, where at the end of the operation v_i will be assigned a random value from its domain. Havoc operations can be used to model non-deterministic values, for example an input provided by the user or the return value of an unknown external function.

The actual state of the program can be described by the program counter (the actual location) and its data (the values of the variables). Therefore, the set of possible states for a program is $C = L \times D_1 \times ... \times D_n$. A *concrete state* $c \in C$ is $c = (l, d_1, ..., d_n)$, which is a location and a value for each variable from its domain.

In the CFA model, each variable is uninitialized at the beginning. Therefore, any state $c = (l_0, d_1, \dots, d_n)$ with the initial location l_0 is considered to be an *initial state* of the program.

A *transition* $c \xrightarrow{\text{op}} c'$ between two concrete states $c = (l, d_1, \dots, d_n)$ and $c' = (l', d'_1, \dots, d'_n)$ exists, if there is an edge $(l, op, l') \in E$ between the locations of the two states with the semantics of the operation *op*.

- If *op* is an assumption $[\varphi]$, then φ has to hold for $d_1, \dots, d_n$ and the values do not change, i.e., $d_k = d'_k$ for each k.
- If *op* is an assignment $v_i := \psi$, d'_i will be equal to the result of evaluating ψ, while the other variables will remain unchanged, i.e., $d'_k = d_k$ for each $k \neq i$.
- If *op* is a havoc over v_i, then d'_i can take any value, but the other variables must be unchanged, i.e., $d'_k = d_k$ for each $k \neq i$.

A *concrete path* $c_1 \xrightarrow{op_1} c_2 \xrightarrow{op_2} ... \xrightarrow{op_{n-1}} c_n$ is an alternating sequence of concrete states and transitions.

The states, the initial state and the transitions together define the *state space* of the program.

During software verification, a wide variety of properties can be verified, including overflow, null pointers and indexing out of bounds [4]. In our work, we focus on verifying *assertion* failures in the input programs. These assertions are represented in CFA with a choice: if the condition of the assertion holds, the program moves forward to the next location, but if it does not holds, it goes to a distinguished *error location* denoted by l_e.

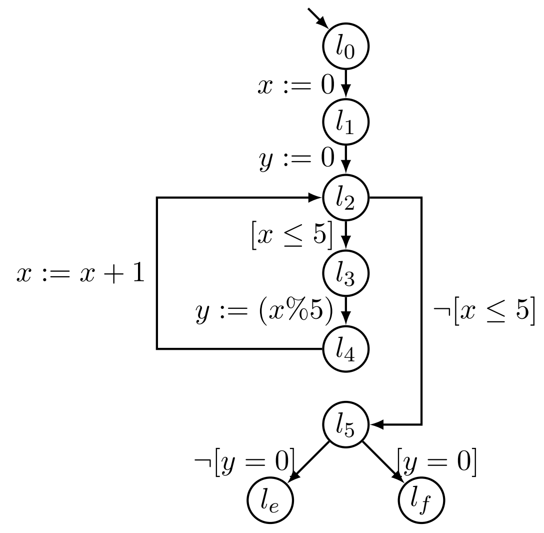
The purpose of *software model checking* [16] is to check (in a mathematically precise way) if a program state with the error location $(l_e, d_1, \dots, d_n)$ is reachable with any valuation of the variables, i.e., whether an assertion failure can occur. Note that this is different than just checking if the error location is reachable in the graph of the CFA. The semantics of the operations also need to be considered. From now on, if we refer to the reachability of l_e, we mean reaching some state in the state space, which has l_e as its location.

A CFA is called *safe* if l_e is not reachable, otherwise it is *unsafe*. If the CFA is unsafe, a path $c_1 \xrightarrow{op_1} c_2 \xrightarrow{op_2} ... \xrightarrow{op_{n-1}} c_n$ leading to the state $c_n = (l_e, \dots)$ with the error location is called a *counterexample*, as it is a witness for the assertion failure. Such counterexamples are important because they help the program developer to identify the source of the problem.

Software model checking is a very complex problem, because if we want to prove that the error location is unreachable, we have to explore the whole state space of the program, which can be very large or even infinite. For example, if a program has 100 locations and three 64 bit integer variables, the number of possible states is $100 \cdot 2^{64} \cdot 2^{64} \cdot 2^{64} \approx 6.2 \cdot 10^{59}$. This problem is often called the "state space explosion". To overcome this limitation of software model checking, various techniques have been developed in the past decades, including abstraction [13, 21] and CEGAR [14], which we present in the next section.

```
int x = 0;
int y = 0;
while (x <= 5) {
  y = x % 5;
  x++;
}
assert(y == 0);
```

(a) Example program.



(b) CFA representation

Figure 2.1: Simple program and its corresponding CFA.

Example 1. *We can see an example program in Figure 2.1a. The program has two variables,* x *and* y*. In the program,* x *counts up to 5 assigning $x\%5$ to* y *in every cycle. At the end there is an assertion which checks whether the value of* y *is 0. In Figure 2.1b, the corresponding CFA can be seen. The initial location is l_0, which is the entry of the program. The first two lines are encoded by path $l_0 \rightarrow l_1 \rightarrow l_2$, where we arrive at the head of the loop. If the condition holds, the program enters the body of the loop by moving to l_3. Then the program moves to l_4 with an assignment and returns to l_2, the head of the loop, incrementing* x*. If the loop condition does not hold anymore, the program moves to l_5, where the assertion is evaluated. If the condition holds, the program arrives to its end, the final location, which is l_f. Otherwise it reaches l_e, the error location.*

2.2 Counterexample-guided abstract refinement

Abstraction is a general method to reduce the complexity of a task by hiding certain details. In the context of software model checking this yields a smaller abstract state space compared to the original (concrete) state space, mitigating the problem of state space explosion. Intuitively, a single abstract state can represent multiple (or even infinite) concrete states [13]. Applying abstraction also means that we lose information, which can lead to incorrect results. However, if we use an *over-approximating* abstraction [13], the incorrect results are only one sided. That means, that if the error location is not reachable in the abstract state space, it is also not reachable in the original state space, i.e., the original program is safe. On the other hand, we might find a counterexample (path leading to the error location) in the abstract state space, which does not exists in the original program. Such counterexamples are called *spurious* and in this case a more precise abstraction is required.

The granularity of the abstraction (i.e., the amount of information hidden) is called the *precision* [7]. For example, a possible abstraction is to omit certain variables from the software and treating them as if they could take any value from their domain. In this case, the precision can be controlled by the amount of variables omitted: fewer omitted variables give more precise abstraction (but possibly larger state space).

Counterexample-Guided Abstraction Refinement (CEGAR) [14] is a widely used technique in software model checking [19, 23, 5, 25], which starts with an initially coarse abstraction to avoid state space explosion. Then it applies refinements iteratively until all spurious counterexamples are eliminated (proving safety) or a real counterexample is found (proving the program to be unsafe).

The steps of a typical CEGAR algorithm [27] can be seen in Figure 2.2. The two main components are the *abstractor* and the *refiner*, whose detailed behaviour will be presented in Section 2.2.1 and Section 2.2.2 respectively. The first step is to build the initial abstraction from the initial (usually coarse) precision, which is done by the abstractor. When a counterexample is found, it is passed to the refiner. If there are no counterexamples, the model is safe due to the over-approximating [13] nature of abstraction. In the next step, the refiner checks whether the counterexample is feasible. If it is feasible, the original model is unsafe. Otherwise, we have a spurious counterexample and the precision of the abstraction is refined, allowing the abstractor to build a more precise (but potentially larger) abstract state space in the next iteration. This process is iterated until there are no abstract counterexamples or a feasible one is found.

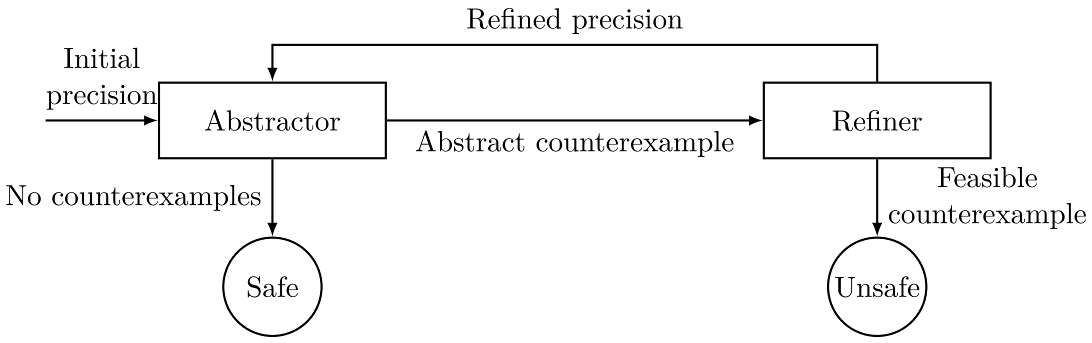


Figure 2.2: CEGAR algorithm

2.2.1 Abstraction

An abstract *domain* is defined by the *abstract states* S, the *coverage relation* $\sqsubseteq$, the set of possible *precisions* Π and the *transfer function* T [7]. Informally, the abstract domain controls the *kind* of information that is hidden to obtain abstract states and the precision defines the *amount* of information hidden. As mentioned previously, a single abstract state can represent any number of concrete states. The coverage relation $s \sqsubseteq s'$ holds for two abstract states $s, s' \in S$, if s' represents all the states that s does. Intuitively, if we already processed s', we can skip s since if the error location is reachable from s, it would have already been reached from s'. The transfer function defines the successor (transition) relation between abstract states.

The abstractor builds the abstract state space (also called an *abstract reachability graph*, ARG [6]) using the parameters above.

Definition 2 (Abstract reachability graph). Formally, an abstract reachability graph is a tuple *ARG = (S, A, C)* where

- S is a set of abstract states from the domain,
- $A \subseteq S \times Ops \times S$ is a set of edges defined by the transfer function T between abstract states, labeled with operations.
- $C \subseteq S \times S$ is a set of covering edges defined by the covering relation $\sqsubseteq$. ▪

The abstractor starts with the initial abstract state, which corresponds to the initial location l_0 and has usually no information as no variable is initialized. Then it maintains a queue for the unprocessed states. As long as the queue is not empty, it picks an abstract state and checks if it can be covered with some already explored state. If yes, it adds the covering edge and leaves the state. Otherwise, it uses the transfer function to calculate its successors. The abstractor stops if there are no more states in the queue or if a state with the error location l_e is found.

In our work, we use three different abstract domains, namely *predicate abstraction* [21], *explicit-value analysis* [5] and their combination, the *product abstraction* [8]. We formalize these domains in the rest of this section.

Explicit-value analysis. Explicit-value analysis is a widely used abstraction method [15, 5, 26]. It tries to reduce the size of the state space by tracking only a subset of the program variables. Usually only a few or no variables are tracked initially and the set of tracked variables is iteratively expanded during the refinement phase. The motivation behind explicit-value analysis is that proving safety (or finding a counterexample) only depends on a small subset of the program variables.

The set of all possible precisions is $\Pi_e = 2^V$, i.e., all possible subsets of the variables. A precision $\pi_e \in \Pi_e$ simply defines the subset of the tracked variables ($\pi_e \subseteq V$), which is also called the set of explicitly tracked variables.

If a variable is not tracked (or unknown), its value is represented by a special *top element* $\top$, meaning that it can take any value from its domain. Given a variable v_i with its domain D_i, let $D_i^\top = D_i \cup \{\top\}$ represent its extension with the top element.

Abstract states $S_e = L \times D_1^\top \times \ldots \times D_n^\top$ track the location and the value of each variable in π_e or $\top$ for variables outside π_e. For example, if there are three variables $V = \{x, y, z\}$ and the precision is $\pi_e = \{x, y\}$, the state $(l_1, 0, 10, \top)$ means that the program is at location l_1, where $x = 0$, $y = 10$ and z is not tracked. Note that it is also possible for a tracked variable to be unknown ($\top$), for example if x is tracked but z is not, the assignment $x := z$ will make $x = \top$.

For each state we calculate its successors with the transfer function $T_e : S_e \times Ops \times \Pi_e \to 2^{S_e}$, which yields a set of successor states for a given state, operation and precision. The values of the tracked variables of the new state depend on the type of operation given to the transfer function.

- If the operation is an assumption $[\varphi]$, we have to check whether it can be evaluated over the source state. If it evaluates to true or can not be evaluated, a successor state is created, where the values of the tracked variables are unchanged. For example, if the assumption is $[x > 5]$ and x is $\top$, the expression cannot be evaluated, therefore we add a successor state, since x can take any value.

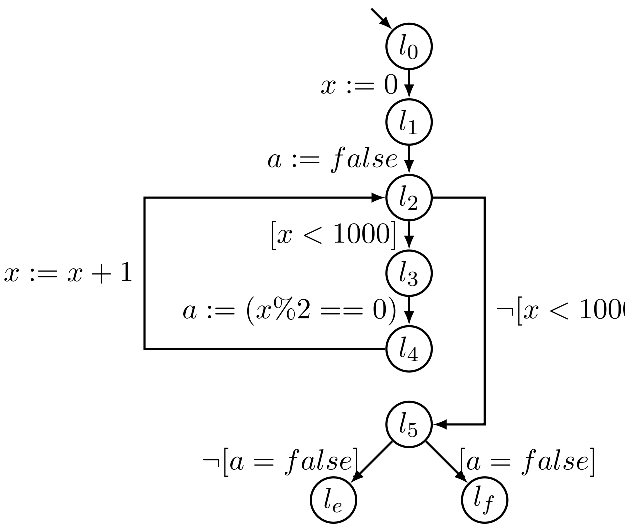
- If the operation is an assignment $v_i := \psi$, a successor can be created. If v_i is not in the set of explicitly tracked variables, the new state's variables are left unchanged. Otherwise, d_i' is assigned the result of evaluating ψ, or if it cannot be evaluated, it is assigned $\top$, while the other variables are unchanged. For example, let the precision be $\pi_e = \{x, y\}$, the source state be $s_e = (l_1, 2, 0, \top)$ and the operation be $op = (x := z + 1)$. Since z is not tracked, the operation cannot be evaluated, therefore the successor abstract state is $(l_1, \top, 0, \top)$.
- If the operation is a havoc, a successor state is created. If the havoced variable is not tracked, the new state's variables are left unchanged. Otherwise, the havoced variable is assigned $\top$ and the other values do not change.

The coverage relation $s \sqsubseteq s'$ holds between two states $s, s' \in S$, if the locations are equal ($l = l'$) and the values of s' are broader than the values of s. This means that if a v_i variable has a value d_i in s, then it must have d_i or $\top$ in s'. For example, $(l_0, 1, 2) \sqsubseteq (l_0, \top, 2)$, but $(l_0, 1, 2) \not\sqsubseteq (l_0, 3, \top)$. Intuitively, if we have a covered state, we do not have to explore the paths starting from this state as they would lead to the same states as the transitions of the covering state.

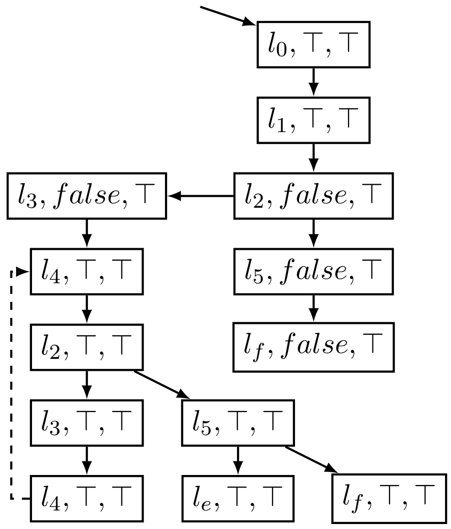
Example 2. *Consider the CFA in Figure 2.3a. It alternates the variable a between true and false while x counts up to 1000. At the end it checks whether a = false. In Figure 2.3b the corresponding ARG with $\pi_e = \{a\}$ can be seen. In this case we only track a, because tracking x while it counts to 1000 would create too many states to explore. The initial state is $(l_0, \top, \top)$, since a is not initialized and x is not tracked (because of this, the value of x will be $\top$ throughout the whole example). The first two transitions set the variables of the program and we arrive to state $(l_2, false, \top)$. Here we are at the head of a loop, whose condition $(x < 1000)$ cannot be evaluated since we do not track* x. *Thus the program has to explore both possibilities. If we do not enter the loop, we move to $(l_5, false, \top)$, where we arrived at the evaluation of the assertion. Since we know that a = false, we can only move to the final location. Otherwise, if we enter the loop, we move to $(l_3, false, \top)$. The next transition is an assignment, which we cannot evaluate because we do not know the value of x. Therefore, we arrive at $(l_4, \top, \top)$. In the next step x is incremented and we move back to l_2, but this time the value of a is unknown. Here we can enter the loop again reaching $(l_3, \top, \top)$ and then $(l_4, \top, \top)$. This is a state where we have been before, so we do not have to explore again. We mark this fact with the dashed covering edge. From $(l_2, \top, \top)$, we can move forward to the end of the loop, $(l_5, \top, \top)$, where we reach the assertion again. This time we can reach both $(l_e, \top, \top)$ and $(l_f, \top, \top)$ since the value of a is unknown.*

Predicate abstraction. In predicate abstraction [21, 14, 1], the values of the variables are not tracked explicitly, but instead certain facts are stored about them. For example, we do not track the exact values of a variable x, but only the fact whether $x < 5$ holds. These fact are called *predicates*, which are Boolean FOL formulas. In case of a too coarse abstraction, refinement is performed by extending the set of tracked predicates.

The set of precisions is $\Pi_p = 2^{FOL^B}$, i.e., all possible subsets of all Boolean formulas. A precision $\pi_p \in \Pi_p$ is a set of Boolean FOL formulas ($\pi_p \subseteq FOL^B$) that are currently tracked.



(a) Example CFA



(b) Corresponding ARG

Figure 2.3: CFA an its corresponding ARG created with $\pi_e = \{a\}$

Abstract states $S_p = L \times 2^{FOL^B}$ are also subsets of predicates with the additional location component. An abstract state $s_p \in S_p$ for the actual precision π_p can contain each predicate of π_p ponated or negated. It is also possible that a predicate does not occur in an abstract state. Then, it represents both cases (it can hold or not). For example, if we have the precision $\pi_p = \{x < 5, y \geq 7\}$, the abstract state $(l_1, \neg(x < 5))$ means that the program is at location l_1, where the negated form of $x < 5$ holds and $y \geq 7$ can both hold or not.

An abstract state represents all concrete states for which the predicates evaluate to true. For example, $(l_1, \neg(x < 5))$ represents states with location l_1, $x = \{4, 3, 2, 1, 0, -1, \ldots\}$ and any value for the other variables.

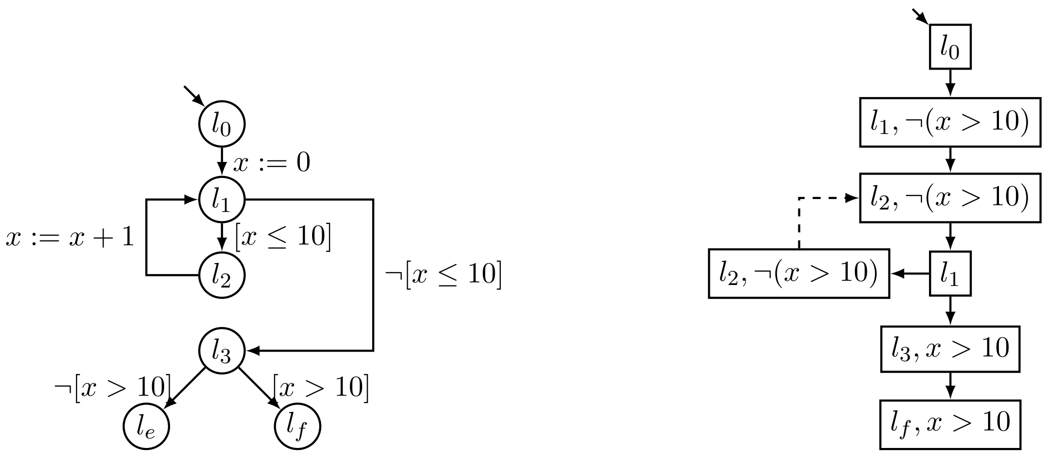
For each state we calculate its successors with the transfer function $T_p : S_p \times Ops \times \Pi_p \to 2^{S_p}$. It works similarly to the transfer function of the explicit-value analysis.

- If the operation is an assumption $[\varphi]$, we check whether the conjunction of the predicates of the source state and φ is feasible. If yes, a successor state is created. The successor state will have all predicates from the precision that are implied by the source state and the assumption. Similarly, if their negation is implied, the successor state will include the negated version. For example, if the source state has the predicate $x \geq 0$ and the assumption is $[x < 0]$, there will be no successor state. On the other hand, if the assumption is $[x \leq 0]$, then a successor state is possible. If there is a predicate $x \neq 0$ in the precision, the successor state can include its negation, as $x \geq 0$ and $x \leq 0$ together imply that $\neg(x \neq 0)$.
- If the operation is an assignment $v_i := \psi$, a new state is created. Similarly to the assumptions, we check whether the predicates of the source state and the assignment implies predicates in the precision or their negated form. For example, let the precision be $\pi_p = \{(x < 5), (y \geq x)\}$, the abstract state be $s_p = (l_1, (x < 5), \neg(y \geq x))$ and the operation be $op = (x := x + 1)$. Incrementing x means that $(x < 5)$ might hold or not, since the value of x can reach 5. On the other hand, it also means that the predicate $\neg(y \geq x)$ holds, because increasing x does not change the fact, that y is less than x. Therefore, the new state is $(l_2, \neg(y \geq x))$.

- If the operation is a havoc, a successor state is created. If the havoced variable appears in a predicate, that predicate will be lost in the target state. Otherwise, the predicates are left unchanged.

The covering relation $s \sqsubseteq s'$ holds for two states if the locations are equal ($l = l'$) and the predicates of s imply the predicates of s'. For example, $(x < 4)$ implies that $(x < 5)$, hence, if we already explored all states from $(x < 5)$, then it already covers all possibilities from $(x < 4)$ as well.

Example 3. *An example for predicate abstraction can be seen in Figure 2.4. In Figure 2.4a, there is an example CFA with a variable x, which counts up to 11 and checks whether its value is greater than 10. In Figure 2.4b, an abstract state space created with precision $\pi_p = \{(x > 10)\}$ can be seen. In the initial state l_0, x has not been initialized yet, therefore we cannot decide if the predicate is true or false. In the next step x is assigned the value 0, hence the $\neg(x > 10)$ predicate holds at l_1. The program then arrives to a selection, and because of the predicate, the program can only move to l_2. Since no assignment happened, the $\neg(x > 10)$ predicate still holds. In the next step x is incremented while returning to l_1, therefore we can not evaluate the predicate anymore. That is why the program can move to both ways from here. If $x \leq 10$, it reaches the state $(l_2, \neg(x > 10))$, which is covered as it already appeared. Otherwise it moves to $(l_3, x > 10)$. At l_3, the program arrives to the evaluation of the assertion. It can only go to the final location from here, because the predicate $x > 10$ stands. Therefore the CFA is safe.*

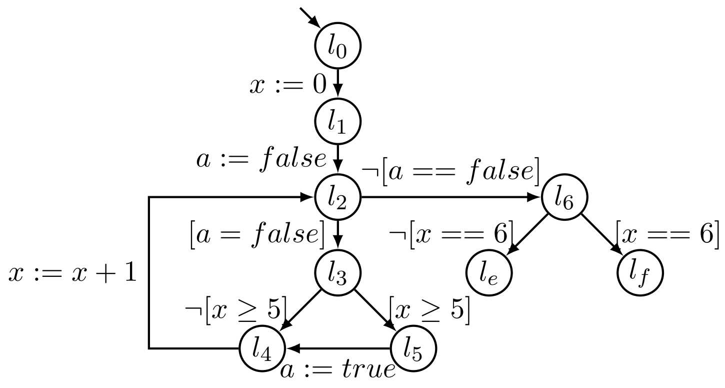


(a) Example CFA **(b)** Corresponding ARG

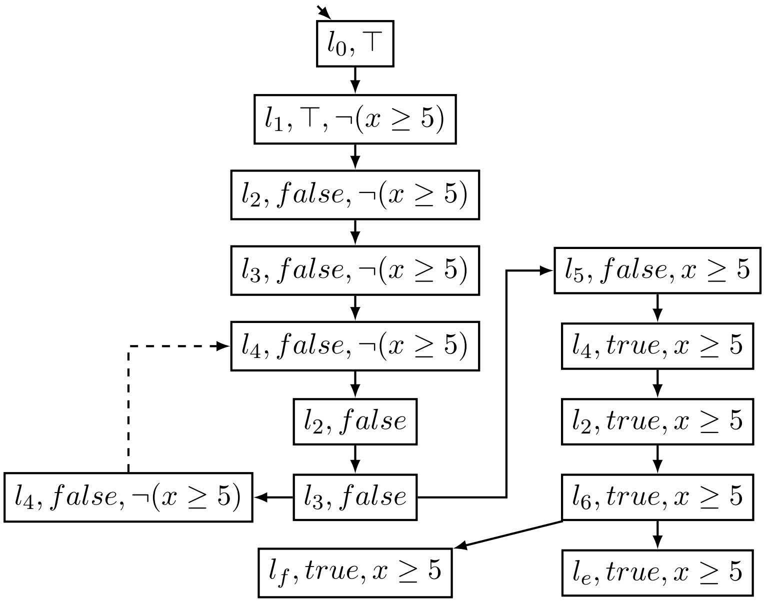
Figure 2.4: CFA an its corresponding ARG created with the precision $\pi_p = \{(x > 10)\}$

Product abstraction. Product abstractions [8] combine different abstract domains. In our case we use a combination of explicit-value analysis and predicate abstraction. The precision is $\Pi = \Pi_p \times \Pi_e$, the combination of the two precisions. An abstract state $S = L \times 2^{FOL^B} \times D_1^\top \times \ldots \times D_n^\top$ consists of a location and of predicates and explicitly tracked variables at the same time. The transfer function $T : S \times Ops \times \Pi \to 2^S$ gets a product state $s = (l, s_p, s_e)$, an operation op, a precision $\pi = (\pi_p, \pi_e)$ and calculates $T_p((l, s_p), op, \pi_p) \times T_e((l, s_e), op, \pi_e)$, that is the Descartes product of the successor predicate states and successor explicit states. A state $(l, s_p, s_e) \in S$ is covered by another state $(l', s'_p, s'_e) \in S$ if both components are covered, i.e., $(l, s_p) \sqsubseteq (l', s'_p)$ and $(l, s_e) \sqsubseteq (l', s'_e)$.

The main research question in product abstraction is how to select which variable gets tracked explicitly and which one gets predicates. We developed multiple strategies for efficiently combining predicates and explicit values. We will explain these in Chapter 3.



(a) Example CFA



(b) Corresponding ARG

Figure 2.5: CFA an its corresponding ARG created with $\pi = \{a, (x \geq 5)\}$

Example 4. *Consider the example CFA in Figure 2.5a. It has two variables, an Integer x and a Boolean a. The program enters a loop and x starts to count up. When it reaches 5, the program assigns a to true, increments x once again and checks whether the value of x is 6. In Figure 2.5b the corresponding abstract state space can be seen, created with precision* $\pi_e = \{a\}$ *and* $\pi_p = \{(x \geq 5)\}$. *Therefore an abstract state consists of the name of the location, the value of a and the predicate* $(x \geq 5)$. *The initial state is* $(l_0, \top, \{\})$ *since the CFA starts at* l_0 *and no variables are initialized. We do not know the value of a and cannot evaluate the predicate. After initializing the variables in path* $l_0 \rightarrow l_1 \rightarrow l_2$, *the program arrives at* $(l_2, false, \neg(x \geq 5))$. *Here it enters the loop, since we know that* $a = false$. *In the next step, the program checks whether* $x \geq 5$ *and moves to* $(l_4, false, \neg(x \geq 5))$. *The next transition increments x and takes the program back to the head of the loop. Since the value of x changed, we cannot evaluate the predicate any more, therefore the program arrives at* (l_2, false), *where it enters the loop again. It moves to* $(l_3, false)$, *where we can now take both paths, either moving back to* $(l_4, false, \neg(x \geq 5))$ *or going to* $(l_5, false, x \geq 5)$.

From here the program moves to $(l_4, true, x \geq 5)$*, changing the value of a to true. The program then goes to* $(l_2, true, x \geq 5)$*, from where it jumps after the loop, arriving at the end of the program,* $(l_6, true, x \geq 5)$*. It checks the assertion, where we can reach both the final and error locations.*

2.2.2 Refinement

The refiner's task is to check whether the abstract counterexample is feasible, and if not, it has to find a new precision. It checks the counterexample by translating the alternating sequence of states and actions into FOL formulas and giving them to an SMT solver [28, 11]. If the SMT solver can find a satisfying assignment, it corresponds to a concrete counterexample.

Otherwise, the counterexample is spurious and the refiner extracts the reason of infeasibility using for example interpolation [24, 29, 22] or unsat cores [23]. This will yield new variables or new predicates that should be tracked. In our work, we treat the refinement as a black-box, which gives a new precision based on the path (counterexample) that should be joined to the old precision.

- In explicit-value analysis, the refiner $R_e\colon \mathit{PATH} \mapsto \Pi_e$ gives new variables to be tracked.
- In predicate abstraction the refiner $R_p\colon \mathit{PATH} \mapsto \Pi_p$ gives new predicates to be tracked.
- In the product abstraction the refiner $R\colon \mathit{PATH} \mapsto \Pi_p \times \Pi_e$ can call both components R_e and R_p and decide which variables and predicates to include. We developed different strategies for this decision, which we present in Section 3.

Chapter 3

Product abstraction strategies

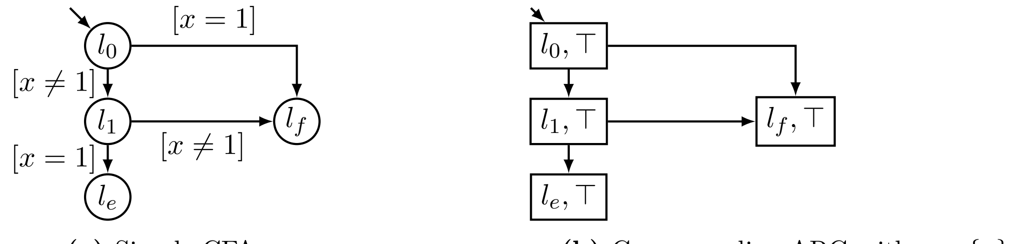
In this chapter we present three different approaches for combining explicit-value analysis and predicate abstraction into a product abstraction. When combining the two different abstractions, a very important question arises: How to decide the new precision when refining the abstraction? The new precision can include new predicates or we can extend the set of explicitly tracked variables, or we can also combine these two methods by picking a predicate for a variable but adding another to the explicitly tracked set.

The main principle is the same for all three strategies: at first, we add a new variable to the explicitly tracked variable set. Our motivation was that handling predicate formulas is more expensive computationally (e.g., checking implications in the transfer function and in the coverage relation). However, the abstract state space might start growing quickly if a variable has a large number of different values and some problems might even not be decidable due to unknown ($\top$) values.

Consider the example CFA in Figure 3.1a, which first checks if $x \neq 1$ and then if $x = 1$ (which obviously cannot be possible). Suppose, that no variables are tracked initially. In this case, the error location is trivially reachable and the refiner extends the precision by adding x. In Figure 3.1b, the corresponding ARG created with explicit-value analysis can be seen. Since x is not initialized the initial state is $(l_0, \top)$. Then we check the condition $x \neq 1$. Since x is unknown, the condition can both hold and not. If it does not hold, the program terminates in the final location l_f. However, if it holds we proceed to l_1, where x is still $\top$. Then we check the condition $x = 1$, which can again hold or not, due to x being unknown. This way, the program can still reach the error location. Since there are no other variables to be tracked, the program cannot be verified (with explicit-value analysis).

A solution for this can be that instead of assigning $\top$ to x, we start to list all the values x can be assigned. It is an effective strategy, if the variable can only have a couple of different values, since knowing the exact values yields more information than a $\top$ value. For example, if the assumption is $0 < x < 5$, then x can only have 4 different values and creating four successor states is still more effective than using a $\top$ value or using predicates. However, in this case this does not solve our problem, since $(x \neq 1)$ means that x can have an infinite number of different values, and this leads to state space explosion.

Our key idea is to introduce a limit k for the number of possible values for each tracked variable. When we cannot evaluate an expression in the transfer function (e.g., $x \neq 1$), we start to enumerate the possible values instead of using a $\top$ value. However, we count the number of different values for each tracked variable and if it is greater than k, we stop



(a) Simple CFA **(b)** Corresponding ARG with $\pi_e : \{x\}$

Figure 3.1: Simple CFA and its corresponding ARG with explicit-value analysis

enumerating and we discard the variable from the explicitly tracked set. We also add this variable to a special *dropouts* set, which is passed over to the refiner.

The strategy of the product refiner R is the following. It calculates both new variables π'_e using R_e and new predicates π'_p using R_p. All variables included in the *dropouts* set are removed from π'_e, since we do not want to track them again. Instead, we (only) keep predicates from π'_p that have a removed variable, other predicates are discarded. In other words, we first always add a variable to the explicit precision. If it is later removed during the transfer function, we do not add it again, but rather add predicates containing that variable.

This way we can (1) avoid working unnecessarily with computationally expensive predicates and (2) we can solve problems that need explicit enumeration instead of a top value, while still avoiding state space explosion. There are multiple ways to count the different values of the variables. In this chapter we present three different approaches.

3.1 Limit number of successors based on a single state

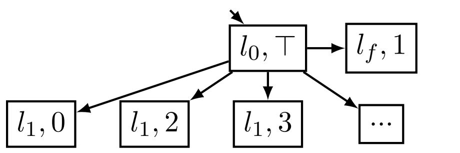
In the first strategy (Algorithm 1), we count the different values of the tracked variables when enumerating successors for a given state. We start by initializing the successor states S'_e as an empty set. We also set a restart flag which will be used later. Then we start enumerating the successor states and we examine the values of the explicitly tracked variables. If the number of the different values of a variable in the successor states exceed k, we add it to the *dropouts* set. We also remove it from π_e, and we set the flag that we should restart the enumeration, since the precision changed (at least one variable was dropped). If there are no more successor states to list and no variable was removed, we do not need to restart and we can return the successor states S'_e.

Example 5. *Consider the example CFA in Figure 3.1a again. If we use the state-based product abstraction, the variable x is added to the set of explicitly tracked variables as previously ($\pi_e = \{x\}$). The corresponding ARG for this precision can be seen in Figure 3.2a. The program starts at state (l_0, T), from where it can go to two different directions. Taking the assumption $[x = 1]$, it arrives at state $(l_f, 1)$ since $x = 1$ is the only possible value satisfying the formula. Otherwise, the program moves to l_1, where it starts to list the possible values for $[x \neq 1]$. There are infinitely many different values, but when we exceed k, the algorithm stops. It removes x from the set of explicitly tracked variables and restarts the enumeration. However, now x is not tracked, so it proceeds to $(l_1, \top)$ without enumerating values and then eventually reaches the error location l_e similarly to Figure 3.1b. The refiner will not add x again since it is included in the dropouts set. Instead, it adds some predicate, e.g., $x = 1$ to the precision π_p. Figure 3.2b shows the ARG created with the new*

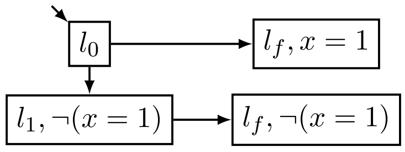
Algorithm 1: State-based transfer function $T_S(s_e, op, \pi_e, k)$.

```
Input  : $s_e$: source state
         $op$: operation
         $\pi_e$: precision
         $k$: bound for successors
Output: $S'_e \subseteq 2^{S_e}$: set of successor states
1  do
2      $S'_e \leftarrow \{\}$
3      restart ← false
4      while new successor state $s'_e$ exists $\wedge\neg$restart do
5          $S'_e \leftarrow S'_e \cup \{s'_e\}$
6          foreach $v_i \in \pi_e$ do
7              if number of different values for $v_i$ in $S' > k$ then
8                  dropouts ← dropouts $\cup \{v_i\}$
9                  $\pi_e \leftarrow \pi_e \setminus \{v_i\}$
10                 restart ← true
11             end
12         end
13     end
14 while restart;
15 return $S'_e$
```

precision. From l_0, the program can arrive to final location $(l_f, x = 1)$ where the predicate is true or move to $(l_1, \neg(x = 1))$ where the negation of the predicate holds. At this point, the predicates keep track that $x \neq 1$ so the algorithm can only proceed to $(l_f, \neg(x = 1))$, where we reached the final location again. Since there are no more states to explore and the algorithm did not reach the error location, the program is safe.



(a) ARG created with precision $\pi_e : \{x\}$



(b) ARG created with precision $\pi_p : \{(x = 1)\}$

Figure 3.2: ARGs created with the state-based strategy

3.2 Limit number of values on a path

The previous strategy only counted different values for the successors of a single state. However, multiple values can occur in other ways as well. For example, if the program includes a loop counting to a large number, then the loop counter i will have a single successor $i + 1$ for each state. However, if we consider the whole path, many different values will start to accumulate: $1, 2, 3, \ldots$.

This example motivated our next strategy, where we examine the number of values of the tracked variables on the path leading to a state when we calculate its successors. Algorithm 2 presents the procedure for this strategy. First we count the number of different values for each variable v_i in the ancestors of s_e (including s_e). If a variable's number of values exceeded the limit k, we add this variable to the *dropouts* set, and remove it from the precision. Then we simply use the original transfer function T_e to calculate the successors, but now some variables might have been removed.

Algorithm 2: Path-based transfer function $T_P(s_e, op, \pi_e, k)$.

```
  Input  : $s_e$: source state
           $op$: operation
           $\pi_e$: target precision
           $k$: bound for successors
  Output: $S'_e \subseteq 2^{S_e}$: set of successor states
1 foreach $v_i \in \pi_e$ do
2   | if number of different values for $v_i$ in ancestors of $s_e > k$ then
3   |   | $dropouts \leftarrow dropouts \cup \{v_i\}$
4   |   | $\pi_e \leftarrow \pi_e \setminus \{v_i\}$
5   | end
6 end
7 $S'_e \leftarrow T_e(s_e, op, \pi_e)$
8 return $S'_e$
```

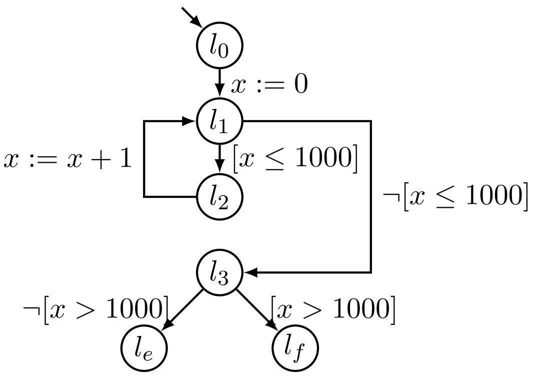
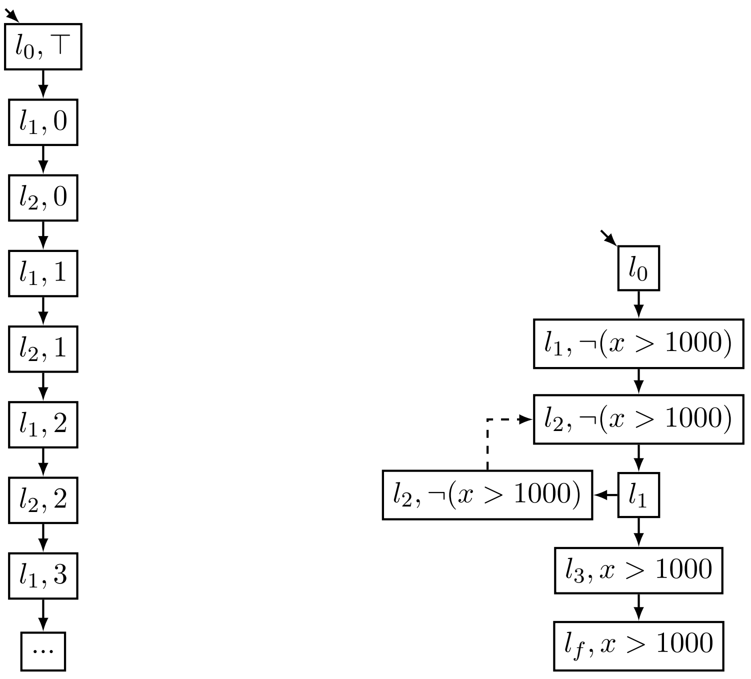


Figure 3.3: Example CFA

Example 6. *Consider the example CFA in Figure 3.3. The program's only variable x counts to 1001, then examines whether its value is greater than 1000. Using path-based product abstraction, x is first added to π_e. When creating the ARG, we arrive at the head of the loop from the initial location. If the program stays in the loop, we get a path, where the value of x is increasing continuously, therefore the number of different values can reach the limit (the corresponding ARG can be seen in Figure 3.4a). When we exceed the limit, we remove x from the set of explicitly tracked variables and instead treat it as a top value. This way the error location can be reached. The refiner will not include x in π_e again, but rather add a predicate, e.g., $x > 1000$ to the precision π_p. The ARG created with the new precision can be seen in Figure 3.4b. The program starts at (l_0), where the predicate cannot be evaluated. After initializing x, it arrives at $(l_1, \neg(x > 1000))$. Because of the predicate, the program moves to $(l_2, \neg(x > 1000))$. In the next step, the value of x is increased, therefore we cannot evaluate the predicate any more, and arrive to (l_1). The program is at the head of the loop again, but now it can go to two different directions. If it enters the loop, it arrives to $(l_2, \neg(x > 1000))$ again. Otherwise it moves to $(l_3, x > 1000)$, from where it arrives at the final location, $(l_f, x > 1000)$. Since there are no more states to explore and the algorithm did not reach the error location, the program is safe. The* *advantage of the path-based approach is that we did not have to explore all* 1001 *values for* x.



(a) ARG created with $\pi : \{x\}$ **(b)** ARG created with $\pi : \{(x > 1000)\}$

Figure 3.4: AGRs created with the path-based strategy

3.3 Limit number of values in ARG

Our third strategy examines the number of the different values in the whole ARG. It generalizes the previous algorithms: it examines the variables in the successor states and also in the previous states through the whole ARG. Algorithm 3 presents the procedure, which is similar to the state-based strategy, except that here we count the number of different values in the whole ARG. Note that in the implementation we use a cache, so that we do not have to traverse the whole ARG at every calculation. Whenever a new state is calculated or a variable is removed, the cache is updated.

3.4 Related work

The combination of different abstract domains have been studied in the literature before. The dynamic precision adjustment approach [8] for the explicit and predicate domains is similar to our ARG-based strategy. The main difference is that our algorithm tries to enumerate states for a formula, while the dynamic precision adjustment method keeps top values.

Refinement selecton [9] focuses on the different refinements returned by the explicit and predicate refiners. Various metrics are defined to compare possible refinements and pick the "better" one. In contrast, our method always tries the explicit refinement, but then switches to predicate if needed.

Algorithm 3: ARG-based transfer function $T_a(s_e, op, \pi_e, k)$.

```
Input  : $s_e$: source state
         $op$: operation
         $\pi_e$: precision
         $k$: bound for successors
Output: $S'_e \subseteq 2^{S_e}$: set of successor states
1  do
2      $S'_e \leftarrow \{\}$
3      restart $\leftarrow$ false
4      while new successor state $s'_e$ exists $\wedge \neg$restart do
5          $S'_e \leftarrow S'_e \cup \{s'_e\}$
6          foreach $v_i \in \pi_e$ do
7              if number of different values for $v_i$ in the ARG $> k$ then
8                  $dropouts \leftarrow dropouts \cup \{v_i\}$
9                  $\pi_e \leftarrow \pi_e \setminus \{v_i\}$
10                 restart $\leftarrow$ true
11             end
12         end
13     end
14 while restart;
15 return $S'_e$
```

Chapter 4

Evaluation

This chapter presents our implementation of the three product abstraction-based strategies and the evaluations of these algorithms including a comparison to explicit-value analysis and predicate abstraction. We ran measurements for every strategy with multiple different k values to examine which k is the most effective for the different algorithms. We then compare the strategies with each other and the two basic algorithms, explicit-value analysis and predicate abstraction.

4.1 Implementation

We implemented the algorithms based on the open source[1] THETA framework [27], which is a modular and configurable model checking framework developed at the Budapest University of Technology and Economics. The explicit-value analysis and predicate abstraction algorithms, and the abstractor and refiner components were already included in THETA [22, 27]. Furthermore, THETA uses Z3 [17] as an SMT solver.

We had to implement the transfer functions for the product abstraction-based strategies and since the refiner has to know which explicitly tracked variables had been removed, we had to modify the refiner as well. We implemented these components in a Gradle[2] Java project, where THETA is imported as a Gradle plug-in.

We also implemented a runnable tool which is deployed in a jar file named `prodanalysis.jar` to run the algorithms with command line arguments. These arguments are given by the following flags.

- `model`: This is a mandatory argument, the path of the CFA file to be checked.
- `domain`: This is the algorithm to run. Its possible values are `EXPL` for explicit-value analysis, `PRED` for predicate abstraction and `PROD2` for product abstraction. It is also a mandatory argument.
- `prodstrategy`: This is the strategy to run product abstraction with. Possible values: `STATE` for state-based, `PATH` for path-based and `ARG` for ARG-based.
- `limit`: This is the limit k for product abstraction. It is an optional parameter with a default value of 5.

[1]`https://github.com/FTSRG/theta`
[2]`https://gradle.org/`

For example, the following call checks `example.cfa` with the state-based product abstraction strategy, where $k = 2$: `java -jar prodanalysis.jar -model example.cfa -domain PROD2 -prodstrategy STATE -limit 2`.

4.2 Measurement configuration

We ran the measurements on a 64 bit Ubuntu 16.04 operating system, with the tool RunExec fron the BenchExec suite [10]. RunExec ensures highly accurate results, since it measures the actual time spent on the CPU and also takes various side-effects into consideration (e.g., memory swapping). BenchExec is also used at the Competition on Software Verification (SV-Comp) [3].

We evaluated 430 input programs from four different sources and categories: plc, eca, locks, ssh. The 90 programs in plc are industrial programmable logic controller (PLC) codes from CERN [20], while the other three categories come from the Competition on Software Verification (SV-Comp) [3, 4]. The category eca contains 180 programs, which describe large event-driven systems, where the events are represented with non-deterministic variables. The category locks contains 143 programs with small locking mechanisms described with non-deterministic integers and if-then-else constructs. The programs in category ssh describe 17 large server-client systems.

We evaluated these programs with 14 different configurations: PRED, EXPL and PROD2 represents the predicate abstraction, the explicit-value analysis and the product abstraction respectively. Behind PROD2, STATE, PATH and ARG represents the different product abstraction strategies presented before, and the number corresponds to the current limit (1, 2, 8 and 32). Thus we have $3 \cdot 4 = 12$ product strategies, and predicate and explicit abstractions giving two more.

We ran every configuration on every model, yielding 6020 measurements. We enforced a time limit of 180 seconds and a memory limit of 4 GB. With this time limit, 3917 measurements terminated successfully. We also checked that the result of the algorithms (safe/unsafe) always correspond to the expected result, increasing our confidence in the soundness of our approaches.

In the following sections, we first evaluate each of the three strategies with different k values and compare them to predicate and explicit analyses. Then we take the best k value for each strategy and compare them to each other. Finally, we also present a summarizing table for all 14 configurations.

4.3 Evaluate different k values for each strategy

In this section we examine the performance of every product abstraction strategy with four different k values. Besides that, they are also compared to PRED and EXPL.

4.3.1 Single state-based strategy

Table 4.1 shows the results of evaluating models with PROD2_STATE with the different limits. The first column shows the configuration, the second represents the number of successful result (i.e., the algorithm terminated) and the third is the total run time in milliseconds. The different configurations are ranked from best to worst. We can see that the PROD2_STATE strategy preforms much better with every limit than the PRED and EXPL algorithms. PROD2_STATE_01 has by far the best results with 354 verified models. We can see that with a larger limit, the performance is worse. Although the difference is small for $k = 2, 8, 32$ (only 1 model).

Configuration	Succ. count	Total time (ms)
PROD2_STATE_01	354	3851763
PROD2_STATE_02	339	3564207
PROD2_STATE_32	338	4178227
PROD2_STATE_08	337	3983408
PRED	325	7279619
EXPL	312	2800263

Table 4.1: Evaluate different k values for STATE

In Figure 4.1, a heatmap can be seen representing the success rate and total time of the configurations in every program category. The greener the tile is, the better the performance. It can be seen that the state-based strategy performs well in general in every category. However, EXPL has the best results in categories eca and ssh, although it has the worst overall performance because of the weak results in category plc. The strategy with $k = 1$ is more successful than other k values due to the plc models.

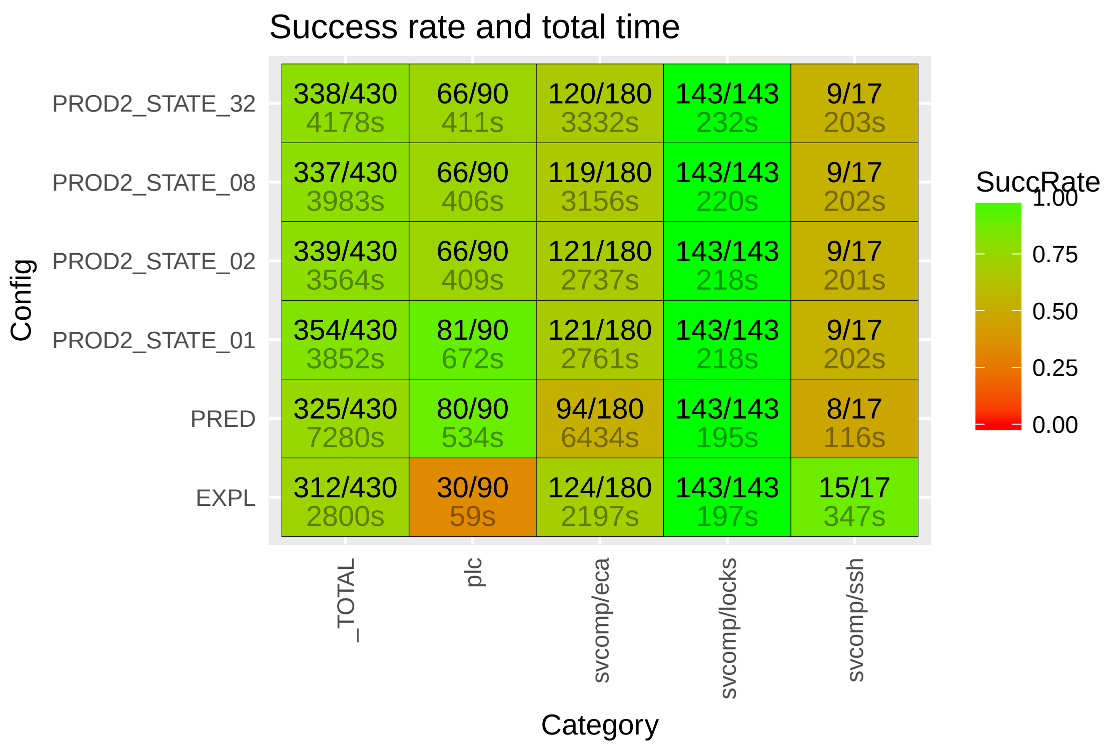


Figure 4.1: Heatmap of the PROD2_STATE strategies

Figure 4.2 represents a quantile plot [10]. It is comparing the maximal time per model on the vertical axis to the number of verified models on the horizontal axis. The performance of the configurations are represented by different coloured lines. A point (x, y) for a given configuration means that it could solve x models within y time for each. Lines shifted to the left could solve more models and lines shifted to the bottom require less time. Therefore, the line located closest to the bottom right corner yields the best performance.

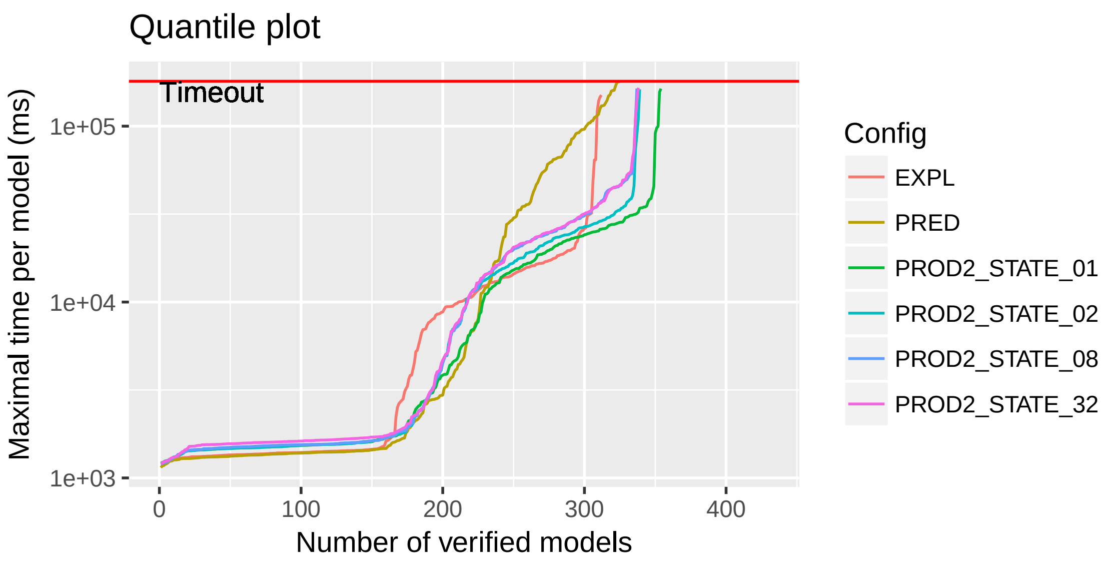


Figure 4.2: Quantile plot of PROD2_STATE

This figure shows the same results as the previous one: PROD2_STATE_01 is the most efficient strategy and PRED and EXPL are the worst, verifying a low number of models with a rather long execution time.

4.3.2 Path-based strategy

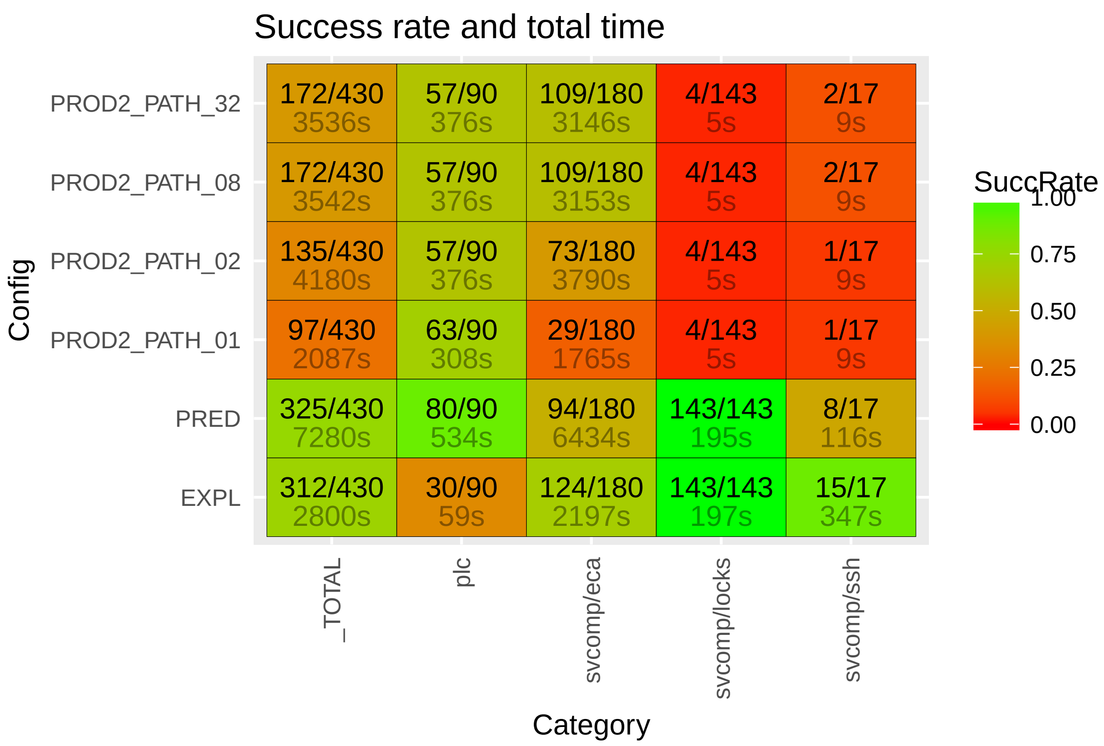
The results of the measurements run with configurations PROD2_PATH, PRED and EXPL can be seen in Table 4.2. We can see, that the PATH strategy has the worst results, verifying almost only half the models the PRED algorithm did. With limits 32 and 8, PATH had the same results, evaluating 172 models. But the smaller the limit is, the worse the performance. PATH could only verify 97 models with limit 1, resulting in the worst overall performance.

Configuration	Succ. count	Total time (ms)
PRED	325	7279619
EXPL	312	2800263
PROD2_PATH_32	172	3535678
PROD2_PATH_08	172	3542355
PROD2_PATH_02	135	4180053
PROD2_PATH_01	97	2087215

Table 4.2: Evaluate different k values for PATH

Figure 4.3 shows a heatmap representing the success rate and total time of the PROD2_PATH strategy and PRED and EXPL algorithms in the different program categories. There is not one category where PROD2_PATH has the best result with any limits. PROD2_PATH_01 performed especially bad in the eca category, resulting in only 29 verified models from 180. A possible reason for this is that in the path-based strategy we use the non-enumerating transfer function. A more detailed evaluation is left for future work.

In Figure 4.6 the quantile plot representing the number of verified models and the maximal time per model of the PATH strategies can be seen. It shows that PRED and EXPL have



Config	_TOTAL	plc	svcomp/eca	svcomp/locks	svcomp/ssh
PROD2_PATH_32	172/430 3536s	57/90 376s	109/180 3146s	4/143 5s	2/17 9s
PROD2_PATH_08	172/430 3542s	57/90 376s	109/180 3153s	4/143 5s	2/17 9s
PROD2_PATH_02	135/430 4180s	57/90 376s	73/180 3790s	4/143 5s	1/17 9s
PROD2_PATH_01	97/430 2087s	63/90 308s	29/180 1765s	4/143 5s	1/17 9s
PRED	325/430 7280s	80/90 534s	94/180 6434s	143/143 195s	8/17 116s
EXPL	312/430 2800s	30/90 59s	124/180 2197s	143/143 197s	15/17 347s

Figure 4.3: Heatmap of the PROD2_PATH strategies

better performance, and we can clearly see that the results of PATH get worse with decreasing the limit.

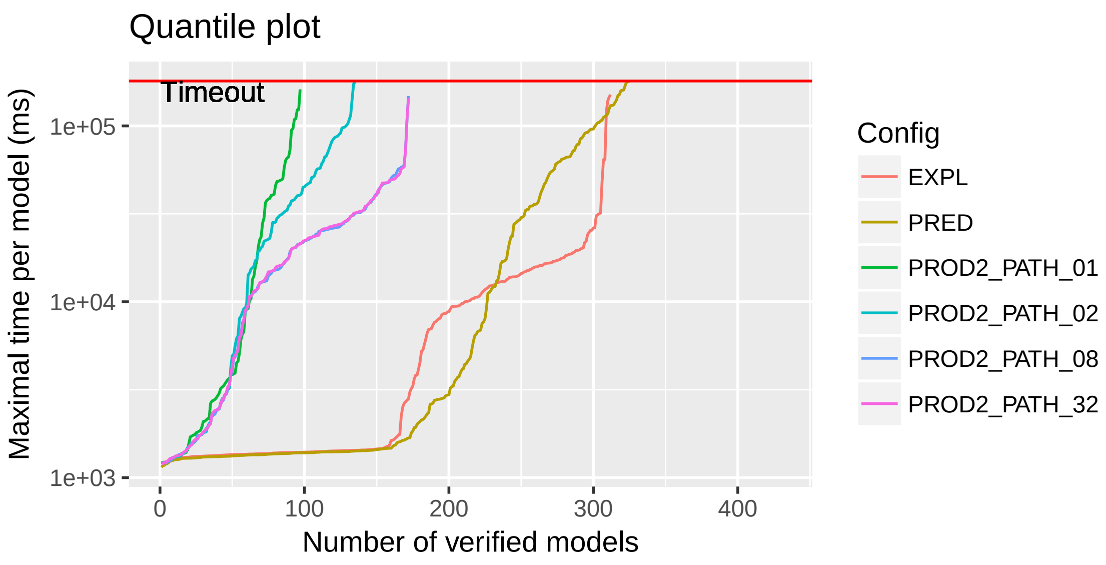


Figure 4.4: Quantile plot of PROD2_PATH

4.3.3 ARG-based strategy

The results of evaluating PROD2_ARG can be seen in Table 4.3. It performed better with all four limits than the PRED and EXPL algorithms. We can see, that the bigger limit yields better performance, resulting in PROD2_ARG_32 being the most effective strategy, verifying 338 models. There is not much difference between the number of models verified, but the execution times vary greatly. PROD2_ARG_01 not only verified the least models of the ARG strategies, but it took more than twice as long.

Config	Succ. count	Total time (ms)
PROD2_ARG_32	338	4169638
PROD2_ARG_08	337	3958952
PROD2_ARG_02	334	5575245
PROD2_ARG_01	327	8598624
PRED	325	7279619
EXPL	312	2800263

Table 4.3: Evaluate different k values for ARG

In Figure 4.5 we can see a heatmap representing the success rate and total time of the PROD2_ARG strategies for every program category. The ARG strategy has good results in every category. It is interesting to note that in the plc category, PROD2_ARG_01 verified remarkably more models, but because of the bad results in category eca, this configuration has the worst performance.

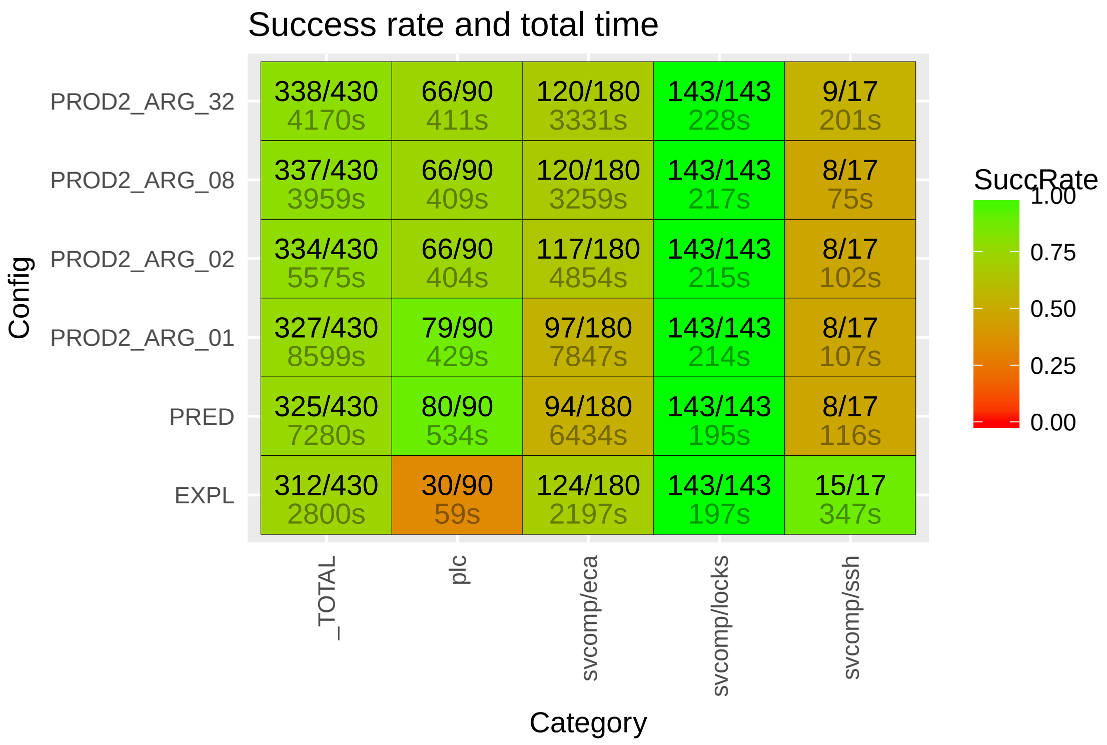


Figure 4.5: Heatmap of the PROD2_ARG strategies

Figure 4.6 represents the quantile plot for PROD2_ARG strategies. Interestingly, there is a part in the plot where EXPL is under all the other algorithms, which means that it had a better performance. But later it could not solve as much models as the others, therefore it is not as efficient.

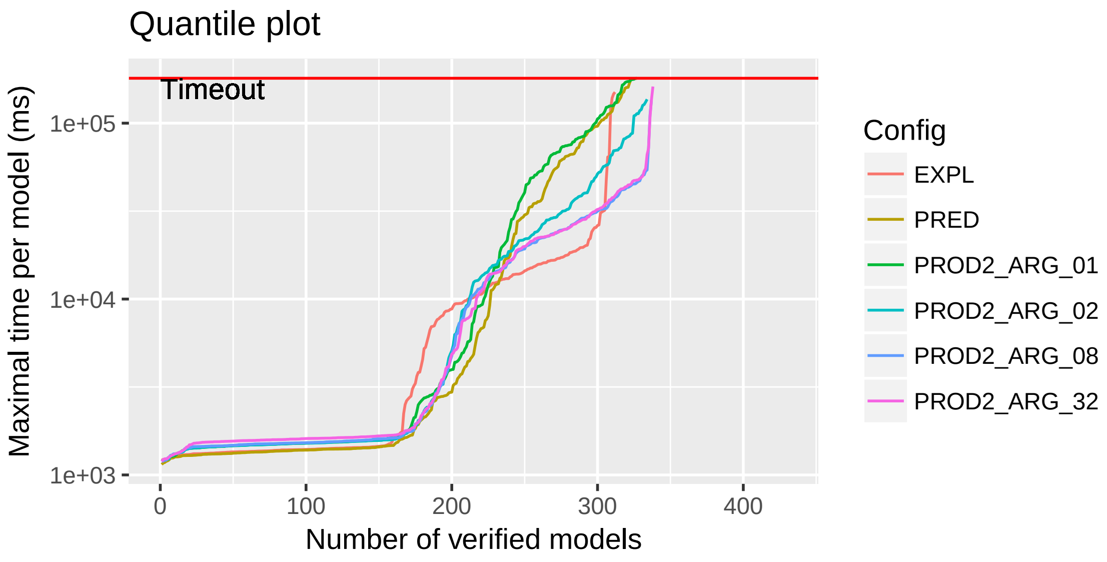


Figure 4.6: Quantile plot of PROD2_ARG

4.4 Compare the best strategies

In this section we compare the configuration with best k value of each strategy with the PRED and EXPL algorithms. These are respectively the PROD2_STATE_01, the PROD2_PATH_32 and the PROD2_ARG_32 configurations.

Table 4.4 presents the results of the best strategies. It shows, that PROD2_STATE_01 has the best results overall. It verified 354 models in less time, than the second best PROD2_ARG_32, which verified 338. PROD2_PATH_32 produced the worst result of the bests with only 172 verified models.

Configuration	Succ. count	Total time (ms)
PROD2_STATE_01	354	3851763
PROD2_ARG_32	338	4169638
PRED	325	7279619
EXPL	312	2800263
PROD2_PATH_32	172	3535678

Table 4.4: Comparing the strategies with their best k value

The heatmap of the best configurations can be seen in Figure 4.7. In the category locks, every strategy and algorithm successfully verified all the models, while the best path-based could only verify 4 out of 143. PROD2_STATE_01 has a an overall good performance in every category. PRED and EXPL also have good results, but there is one category for each of them, where their performance is really weak (eca for PRED and plc for EXPL). It can be clearly seen that two of our new strategies (state- and ARG-based) can combine the advantages of explicit value-analysis and predicate abstraction to give an overall better performance.

The quantile plot of the best strategies can be seen in Figure 4.8. We can see that the least efficient strategy is by far the PATH. Not only it verified the least models, it also took the longest time. PRED and EXPL started great, but they evaluated less models than

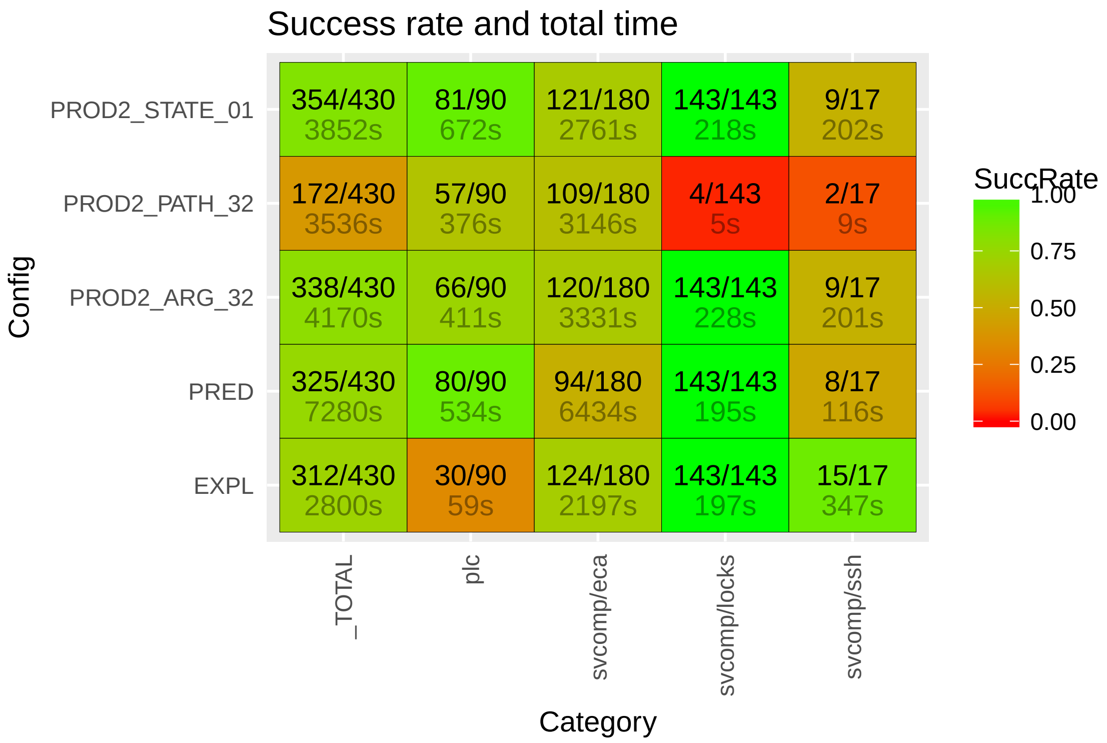


Figure 4.7: Heatmap of the best strategies

ARG and STATE, and PRED is well above the others at the end, meaning it took more time.

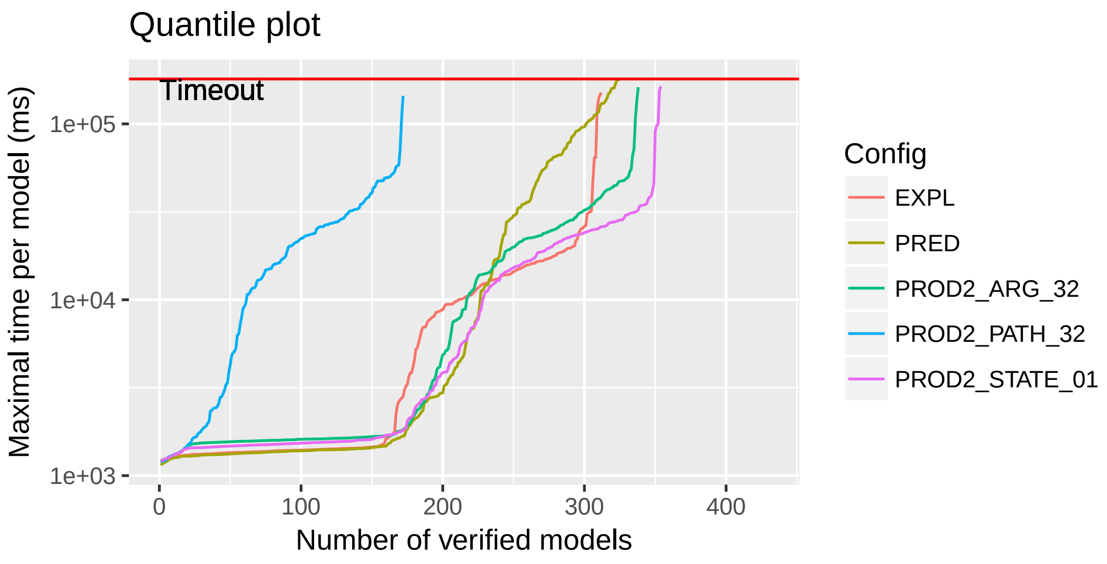


Figure 4.8: Quantile plot of the best strategies

4.5 Compare all strategies with each other

Table 4.5 summarizes the results of every configuration starting from the best to the worst. PROD2_STATE_01 produced by far the best results with 354 successfully evaluated models with a good total time. We can see that PROD2_STATE with a low limit is the most effective configuration. Then comes PROD2_ARG and PROD2_STATE with the same results for limits 32 and 8 not far behind each other. PROD2_ARG follows with a low limit, and then PRED. It is important to note, that evaluating 325 and 327 models with PRED and PROD2_ARG_01 took more than twice as long (not counting timeouts) as evaluating 354 models with the best configuration. The next row shows EXPL with 312 evaluated models. At the end of the table, the PROD2_PATH algorithms can be seen. They had the worst performance, evaluating less than half the models the best configuration evaluated.

Configuration	Succ. count	Total time (ms)
PROD2_STATE_01	354	3851763
PROD2_STATE_02	339	3564207
PROD2_ARG_32	338	4169638
PROD2_STATE_32	338	4178227
PROD2_ARG_08	337	3958952
PROD2_STATE_08	337	3983408
PROD2_ARG_02	334	5575245
PROD2_ARG_01	327	8598624
PRED	325	7279619
EXPL	312	2800263
PROD2_PATH_32	172	3535678
PROD2_PATH_08	172	3542355
PROD2_PATH_02	135	4180053
PROD2_PATH_01	97	2087215

Table 4.5: Comparing all strategies

From these results we can conclude that the state- and ARG-based strategies can successfully combine the strengths of explicit-value analysis and predicate abstraction in order to give an overall better performance.

Chapter 5

Conclusion

In our work, we presented two different CEGAR-based algorithms for software model checking, namely explicit-value analysis and predicate abstraction. Explicit-value analysis only tracks the values of a subset of program variables, while predicate abstraction focuses on tracking formulas over the variables. Both methods can be suitable for checking different kinds of software. In order to combine their advantages, we proposed a product abstraction domain with three different strategies. These approaches start by explicitly tracking each variable first and then switch to predicate abstraction, if the number of different values for a variable exceed a given limit. The difference between the methods is the way they count the values. Counting can be based on a single state, a path or the whole abstract reachability graph.

We implemented our new strategies based on the open source THETA verification framework. We ran measurements on various input programs and compared the strategies to each other and the two basic algorithms (explicit values and predicates). We used benchmark models from the Software Verification Competition and industrial codes from CERN. Measurements show, that the path-based strategy is less efficient, but the state- and ARG-based strategies both outperform pure explicit-value analysis and predicate abstraction. We can conclude that our new algorithms can successfully combine the advantages of the different abstract domains, providing a more efficient software model checking approach.

Future work. Even though the evaluation confirmed the efficiency of the new strategies, there are several opportunities to improve our work.

In the path-based strategy, we are using the original explicit transfer function to calculate the successors. However, this transfer function does not enumerate all possible states if there is more than one possible value of a variable, but assigns the top value to that variable. We could implement this strategy like the others, i.e., to enumerate all the possible successor states, and when the number of different values of a variable in the successor states and in the path reach the limit, we remove that from the precision.

We could also implement a strategy that first does not enumerate possible values for an explicitly tracked variable, but uses the top value. Then, if the refiner would add this variable again, we would start to enumerate values. Finally, if the number of the different values reach the limit, we switch to predicates.

It would be interesting to run the measurements on a wider set of models, possibly from different domains. This would help to generalize our results. Currently we only experimented with a few values for the limit. Evaluating more possibilities could give further insights. Furthermore, the CEGAR algorithm also has some other parameters (independent from the abstract domains), such as the search strategy in the abstract state space. It would be interesting to experiment with those parameters as well, to find a configuration that works the best with product abstraction.

Bibliography

[1] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.

[2] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS press, 2009.

[3] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *Lecture Notes in Computer Science*, pages 887–904. Springer, 2016.

[4] Dirk Beyer. Software verification with validation of results. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10206 of *Lecture Notes in Computer Science*, pages 331–349. Springer, 2017.

[5] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013.

[6] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.

[7] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[8] Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz. Program analysis with dynamic precision adjustment. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38. IEEE, 2008.

[9] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement selection. In *Model Checking Software*, volume 9232 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2015.

[10] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 2017. Online first.

[11] Dirk Beyer, Matthias Dangl, and Philipp Wendler. A unifying view on SMT-based software verification. *Journal of Automated Reasoning*, 60(3):299–335, 2018.

[12] Aaron R Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification.* Springer, 2007.

[13] Edmund Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[15] Edmund M Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.

[16] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick P Bloem. *Handbook of model checking.* Springer, 2018.

[17] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[18] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[19] Evren Ermis, Jochen Hoenicke, and Andreas Podelski. Splitting via interpolants. In *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2012.

[20] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor M. González Suárez. Applying model checking to industrial-sized PLC programs. *IEEE Trans. on Industrial Informatics*, 11(6):1400–1410, 2015.

[21] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[22] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. In *Formal Techniques for Distributed Objects, Components and Systems*, volume 9688 of *Lecture Notes in Computer Science*, pages 158–174. Springer, 2016.

[23] Martin Leucker, Grigory Markin, and MartinR. Neuhäußer. A new refinement strategy for CEGAR-based industrial model checking. In *Hardware and Software: Verification and Testing*, volume 9434 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2015. DOI: `10.1007/978-3-319-26287-1_10`.

[24] Kenneth L McMillan. Applications of Craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[25] Kenneth L McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[26] Cong Tian, Zhenhua Duan, and Zhao Duan. Making CEGAR more efficient in software model checking. *IEEE Transactions on Software Engineering*, 40(12):1206–1223, 2014.

[27] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179. FMCAD inc., 2017.

[28] Y. Vizel, G. Weissenbacher, and S. Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.

[29] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2009.