MŰEGYETEM 1782

DIPLOMATERVEZÉSI FELADAT

Bajkai Viktória Dorina (HUJK05)
szigorló mérnökinformatikus hallgató részére

Hatékony szoftver modellellenőrzés predikátumabsztrakció és explicit változó analízis kombinálásával

Napjainkban a szoftverek az élet olyan területein is egyre elterjedtebbek, ahol helyes viselkedésük garantálása elengedhetetlen a biztonságos működéshez. Ezeken a területeken különböző verifikációs technikákat alkalmaznak, például szoftver modellellenőrzést, amely a program állapotterének szisztematikus vizsgálatával bizonyos hibák meglétét és hiányát is matematikailag precíz módon tudja bizonyítani. A módszer hátránya azonban a nagy számítási igénye, amely általában meggátolja közvetlen használatát valós méretű programokra.

Az ellenőrzés hatékonyságának növelésére gyakran alkalmazzák az úgynevezett ellenpélda-alapú absztrakció finomítás (CEGAR) módszerét, amely absztrakciók iteratív építésével és finomításával csökkenti a vizsgált állapottér komplexitását. A szakirodalomban számos különböző típusú absztrakciót dolgoztak ki, többek között az explicit változó analízist, amely bizonyos változókat nem vesz figyelembe, illetve a predikátumabsztrakciót, amely konkrét értékek helyett logikai formulák teljesülését vizsgálja. A gyakorlati tapasztalatok alapján a különböző típusú absztrakciók más esetekben hatékonyak, így az irodalomban ezek kombinációi, az úgynevezett szorzat absztarkciók is megjelentek.

A hallgató korábbi – szakdolgozat és önálló laboratórium – munkája során szorzat absztrakciókat dolgozott ki különálló algoritmusok formájában. Ezen munka folytatásaként a hallgató diplomatervezési feladata, hogy ezeket az algoritmusokat egy olyan egységes keretrendszerbe illessze, amely a klasszikus CEGAR-alapú megközelítéseket hatékonyan egészíti ki. Ezáltal lehetővé válik a CEGAR algoritmus többi komponensének hatékony újrafelhasználása és az algoritmusok szisztematikus összehasonlítása is.

A hallgató feladatának a következőkre kell kiterjednie:

- Ismertesse a szoftver-modellellenőrzésben használt explicit változó analízist és predikátumabsztrakciót, mutassa be a szorzat absztrakció és a CEGAR működését!
- Egészítse ki az irodalomból ismert klasszikus CEGAR megközelítést úgy, hogy az képes legyen különböző szorzat absztrakciós stratégiák hatékony megvalósítására.
- Dolgozzon ki további, az irodalomból ismert stratégiákat a különböző absztrakciós módszerek kombinálására!
- A nyílt forráskódú THETA verifikációs keretrendszerben található CEGAR megközelítés kiegészítésével implementálja saját keretrendszerét és algoritmusait!
- Példa szoftverek segítségével demonstrálja az algoritmusok működését és hatékonyságát!

Tanszéki konzulens: Hajdu Ákos, egyetemi tanársegéd

Budapest, 2020. 03. 13.

…………………………..
Dr. Dabóczi Tamás
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

MŰEGYETEM 1782

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

Efficient combinations of predicate abstraction and explicit-value analysis for software model checking

Master's Thesis

Author
Viktória Dorina Bajkai

Advisor
Dr. Ákos Hajdu

December 20, 2020

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Bajkai Viktória Dorina*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. december 20.

Bajkai Viktória Dorina
hallgató

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.

Munkánk 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. Ezen kívül az újonnan javasolt algoritmusokat egy egységes keretrendszerben formalizáltuk és 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 program. 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 also formalize the proposed algorithms in a unified framework and implement our new strategies in THETA, an open source verification framework. This allows us to perform experiments 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 experiments 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 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. 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 level of 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 seven 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.

Besides proposing new strategies, an important contribution of this work is formalizing and implementing these new strategies in a unified framework.

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. First, we present Control Flow Automata (Section 2.1), a formal representation based on graphs and first order logic formulas, which we use to describe the programs. 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. Since software verification is based on mathematical reasoning, a formal representation is required. 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) [13] 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 $x = y \land y > 5$.

While FOL is undecidable in general [13], in practice *satisfiability modulo theory* (SMT) solvers [3, 19] can efficiently reason about FOL formulas under various 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 automaton [8] is a tuple $CFA = (V, L, l_0, E)$ where

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of program variables with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$,
- $L = \{l_1, l_2, \ldots, l_k\}$ 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 control flows 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.

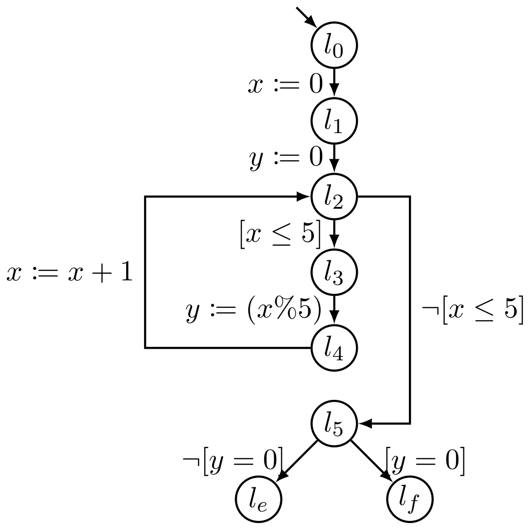
Assignments are in the form $v_i := \psi$, where $v_i \in V$ and $\psi \in FOL^{D_{v_i}}$. After this operation, in the target location v_i will be assigned the result of evaluating ψ. All other variables will have the same value as in the source location.

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.

```
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.*

The *concrete data state* $c \in D_{v_i} \times \dots \times D_{v_n}$ assigns for each variable a value of its domain. A *concrete state* of the program can be described by the program counter (the current location) and its data (the values of the variables), the concrete data state. Therefore, a *concrete state* is $s = (l, c)$, 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 $s = (l_0, c)$ with the initial location l_0 is considered to be an *initial state* of the program.

A *transition* $s \xrightarrow{\text{op}} s'$ between two concrete states $s = (l, c)$ and $s' = (l', c')$ 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 $d_{v_1}, \dots, d_{v_n}$ and the values do not change when taking the transition, i.e., $d_{v_k} = d'_{v_k}$ for each k.

- If *op* is an assignment $v_i := \psi$, d'_{v_i} will be equal to the result of evaluating ψ, while the other variables will remain unchanged, i.e., $d'_{v_k} = d_{v_k}$ for each $k \neq i$.
- If *op* is a havoc over v_i, then d'_{v_k} can take any value, but the other variables must be unchanged, i.e., $d'_{v_k} = d_{v_k}$ for each $k \neq i$.

A *concrete path* $s_1 \xrightarrow{op_1} s_2 \xrightarrow{op_2} ... \xrightarrow{op_{n-1}} s_n$ is an alternating sequence of concrete states and operations.

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 [5]. 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* [17] is to check (in a mathematically precise way) if a program state with the error location $(l_e, d_1, \ldots, d_n)$ 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 $s_1 \xrightarrow{op_1} s_2 \xrightarrow{op_2} ... \xrightarrow{op_{n-1}} s_n$ leading to the state $s_n = (l_e, \ldots)$ 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, the whole state space of the program has to be explored, 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 [14, 22] and CEGAR [15], which will be presented in the next section.

2.2 Counterexample-guided abstraction refinement

Abstraction is a general method to reduce the complexity of a problem 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 [14]. Applying abstraction also means that we lose information, which can lead to incorrect results. However, if we use an *over-approximating* abstraction [14], 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* [8]. 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 number of variables omitted: fewer omitted variables give more precise abstraction (but possibly larger state space).

Counterexample-Guided Abstraction Refinement (CEGAR) [15] is a widely used technique in software model checking [20, 26, 6, 28], which starts with a coarse initial 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 [30, 24] can be seen in Figure 2.2. The two main components are the *abstractor* and the *refiner*, whose detailed behavior 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 [14] 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.

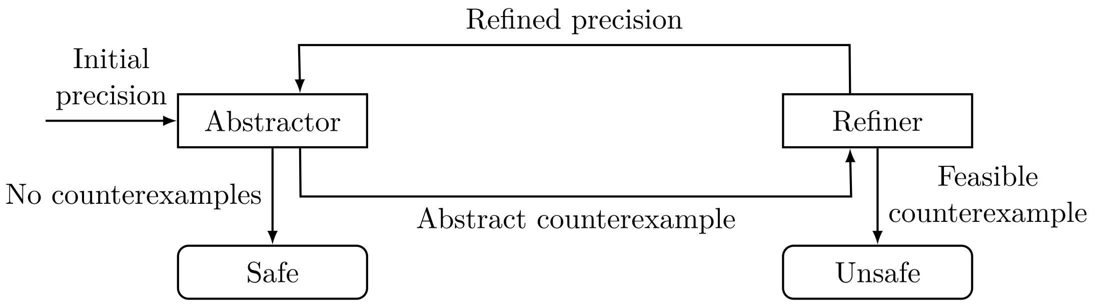


Figure 2.2: CEGAR algorithm

2.2.1 Abstraction

An abstract *domain* is defined by the *abstract nodes* N, the *coverage relation* $\sqsubseteq$, the set of possible *precisions* Π and the *transfer function* T [8]. 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 node can represent any number of concrete states. The coverage relation $s \sqsubseteq s'$ holds for two abstract states $s, s' \in N$, 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 calculates the successor (transition) relation between abstract states.

In our work, we use three different abstract domains, namely *predicate abstraction* [22], *explicit-value analysis* [6] and their combination, the *product abstraction* [9]. We formalize these domains in the rest of this section.

Explicit-value analysis. Explicit-value analysis is a widely used abstraction method [16, 6, 29]. 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 $x = \top$.

An abstract state represents the concrete states where the assignments for the explicitly tracked variables hold. Also in case of a $\top$ value this means every value in its domain. For example, for two variables $V = \{x, y\}$, $(l_1, 0, \top)$ represents states with location l_1, $x = 0$ and any value for the y variable.

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.

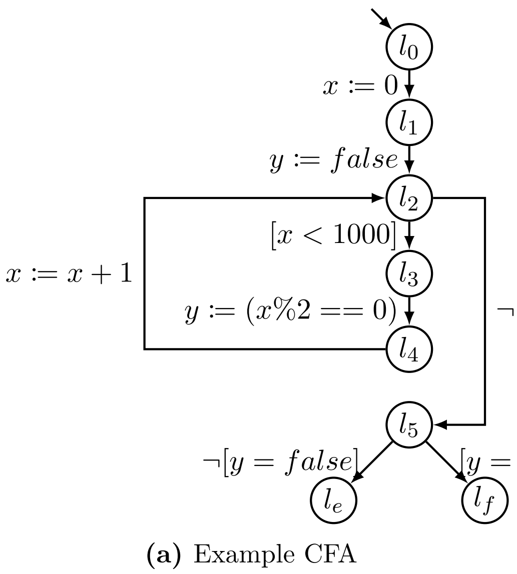
- If the operation is an assignment $v_i := \psi$, a successor can be created. If v_i is not in the set of explicitly tracked variables, the new state's variables are left unchanged. Otherwise, d_i' is assigned the result of evaluating ψ, or if it cannot be evaluated, it is assigned $\top$, while the 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.

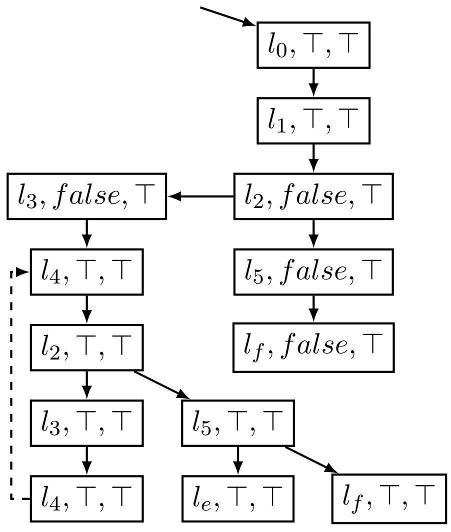
The coverage relation $s \sqsubseteq s'$ holds between two states $s, s' \in S$, if the locations are equal ($l = l'$) and the values of s' are broader than the values of s. This means that if a v_i variable has a value d_i in s, then it must have d_i or $\top$ in s'. For example, $(l_0, 1, 2) \sqsubseteq (l_0, \top, 2)$, 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 paths from the covering state.

Example 2. *Consider the CFA in Figure 2.3a. It alternates the variable y between true and false while x counts up to 1000. At the end it checks whether y = false. In Figure 2.3b the corresponding ARG with* $\pi_e = \{y\}$ *can be seen. In this case we only track y, because tracking x while it counts to 1000 would create too many states to explore. The initial state is* $(l_0, \top, \top)$*, since y 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* $(l_5, \mathit{false}, \top)$*, where we arrived at the evaluation of the assertion. Since we know that y = 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 y 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 y is unknown.*

However, this example code is safe*, but here in this example* l_e *has been reached in the abstract state space. This means, that tracking only variable y explicitly yielded a spurious counterexample and the precision has to be refined by adding a new program variable to the explicitly tracked set.*



(a) Example CFA



(b) Corresponding ARG

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

Predicate abstraction. In predicate abstraction [22, 15, 2], 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 facts 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.

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, while $y \geq 7$ 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.

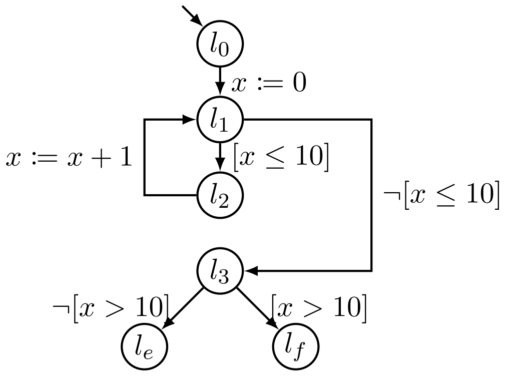
- If the operation is an assumption $[\varphi]$, we check whether the conjunction of the predicates of the source state and φ is feasible. If yes, a successor state is created. The successor state will have all predicates from the precision that are implied by the source state and the assumption. Similarly, if their negation is implied, the successor state will include the negated version [1]. 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 successor 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 successor 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.

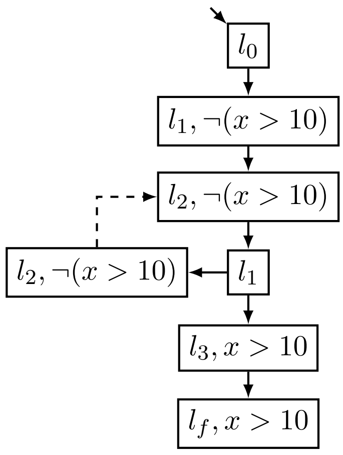
Example 3. *An example for predicate abstraction can be seen in Figure 2.4. In Figure 2.4a, there is an example CFA with a variable x, which counts up to 11 and checks*

[1]This method described is called *Cartesian predicate abstraction* [24]. An other predicate abstraction method is the *Boolean*, where an arbitrary boolean combination of the predicates can be included in the states.

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. This is a great example of how a simple predicate creating a compact abstraction can prove the correctness of a program, without having to traverse the whole concrete state space.



(a) Example CFA



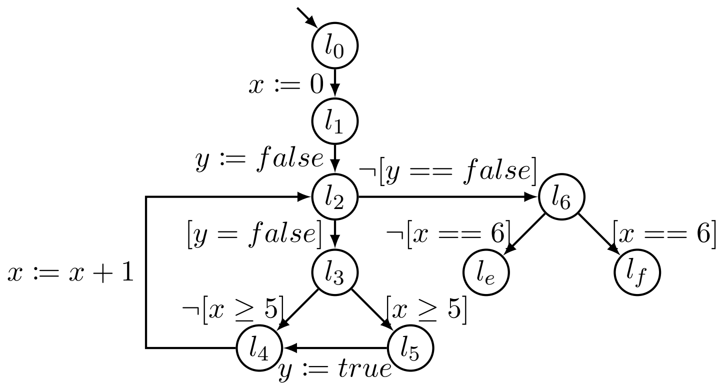
(b) Corresponding ARG

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

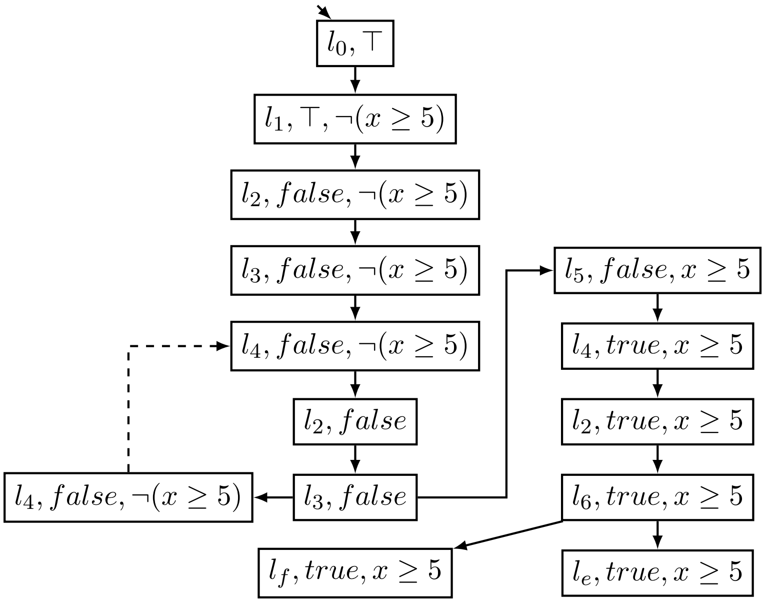
Product abstraction. Product abstractions [9] combine different abstract domains. In our case we use a combination of explicit-value analysis and predicate abstraction. The precision is $\Pi = \Pi_p \times \Pi_e$, the combination of the two precisions. An abstract state $S = L \times 2^{FOL^B} \times D_1^\top \times \ldots \times D_n^\top$ consists of a location and of predicates and explicitly tracked variables at the same time. The transfer function $T : S \times Ops \times \Pi \rightarrow 2^S$ 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.

Example 4. *Consider the example CFA in Figure 2.5a. It has two variables, an integer x and a Boolean y. The program enters a loop and x starts to count up. When it reaches 5, the program assigns y 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 $\pi_e = \{y\}$ and $\pi_p = \{(x \geq 5)\}$. Therefore an abstract state consists of the name of the*



(a) Example CFA



(b) Corresponding ARG

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

location, the value of y 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 y 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* $y = false$*. 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 y 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.*

The abstractor builds the abstract state space (also called an *abstract reachability graph*, ARG [7]) using the parameters above.

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

- N is a set of abstract states from the domain,
- $A \subseteq S \times Ops \times S$ is a set of edges (actions) 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 $\sqsubseteq$. ▪

Algorithm 1: Abstraction algorithm. [23]

```
Input  : $ARG = (N, E, C)$: partially constructed abstract reachability graph
         $l_E$: error location
         $D_L = (S_L, \bot_L, \sqsubseteq_L, \mathsf{expr}_L)$: abstract domain with locations
         $\pi$: current precision
         $T_L$: transfer function with locations
Output: (safe or unsafe, $ARG$)
waitlist := unmarked nodes from $N$
while waitlist $\neq \emptyset$ do
    $l, s$ := remove from waitlist
    // Check if $(l, s)$ is unsafe
    if $l = l_E$ then
        return (unsafe, $ARG$)
    end
    // Check if $(l, s)$ can be covered
    else if $\exists (l', s') \in N : (l, s) \sqsubseteq_L (l', s')$ then
        $C := C \cup \{(l, s, l', s')\}$ // Add covered-by edge
    end
    // Otherwise $(l, s)$ gets expanded
    else
        foreach $(l', s') \in T_L((l, s), \pi) \setminus \bot_L$ do
            waitlist := waitlist $\cup \{(l', s')\}$
            $N := N \cup \{(l', s')\}$ // Add new node
            $E := E \cup \{(l, s, op, l', s')\}$ // Add successor edge
        end
    end
end
return (safe, $ARG$)
```

Algorithm 1 presents the abstraction process used in this Thesis. The input is a partially constructed ARG, an error location l_E, an abstract domain D_L with location, a current precision π_L and a transfer function T_L with locations.

At first, the abstractor starts with the initial abstract state and precision. The initial abstract state correspond to the initial location l_0 and usually has no information since no variable has been initialized yet. Although an initial precision can be given to the abstractor, it is usually empty, therefore at the beginning no variables or predicated are tracked.

The first step of the algorithm is to collect the unmarked states of N into a *waitlist.* An abstract state is *unmarked*, if it has not been expanded nor covered, and neither contains an error location. In other words, it has not been processed yet by the algorithm. The next step is to remove and process states from the waitlist. If the currently examined state contains the error location, the algorithm terminates and returns with an unsafe result. Otherwise it is checked if the state can be covered with al already reached state. If so, a covering edge is added, and if not, the state gets expanded. Expanding a state means calculating its successor states, which is done by the transfer function. These successor states are then added to the waitlist. When there are no more nodes left to examine and no error location was found, the abstraction returns with a safe result and the created ARG.

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 [31, 12]. 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 [27, 32, 25] or unsat cores [26]. This will yield new variables or new predicates that should be tracked.

The refinement algorithm of which this our work was based on is presented in 2. The refiner gets an unsafe ARG, an error location and the current precision as inputs, and returns an unsafe or spurious result with the adjusted precision and ARG.

Algorithm 2: Refinement algorithm. [23]

```
Input  : $ARG = (N, E, C)$: unsafe abstract reachability graph
         $l_E$: error location
         $\pi$: current precision
Output: (unsafe or spurious, $\pi'$, $ARG$)
1  $\sigma = ((l_1, s_1), op_1, \ldots, op_{n-1}, (l_n, s_n)) :=$ path to unsafe node (with $l_E$) from $ARG$
2  // Feasibility check
3  if $s_1^{\langle 1 \rangle} \land op_1^{\langle 1 \rangle} \land \ldots \land op_{n-1}^{\langle n-1 \rangle} \land s_n^{\langle n \rangle}$ is satisfiable then return (unsafe, $\pi$, $ARG$);
4  else
5      $(I_1, \ldots, I_n) :=$ get interpolant for $\sigma$
6      // Precision adjustment
7      $(\pi_1, \ldots, \pi_n) :=$ map interpolant $(I_1, \ldots, I_n)$ to precisions
8      $\pi' := \pi$
9      // Pruning
10     $i :=$ lowest index for which $I_i \notin \{true, false\}$
11     $N_i :=$ all nodes in the subtree rooted at $(l_i, s_i)$
12     $N := N \setminus N_i$ // Prune nodes
13     $E := \{(n_1, op, n_2) \in E \mid n_1 \notin N_i \land n_2 \notin N_i\}$ // Prune successor edges
14     $C := \{(n_1, n_2) \in C \mid n_1 \notin N_i \land n_2 \notin N_i\}$ // Prune covered-by edges
15     return (spurious, $\pi'$, $ARG$)
16 end
```

The first step is to extract the path $\sigma = ((l_1, s_1), op_1, \ldots, op_{n-1}, (l_n, s_n))$ to the unsafe state containing l_E, then its feasibility is checked. If the path is feasible, the counterexample is real, therefore the program is unsafe and the refinement terminates. Otherwise an interpolant [27, 24] is calculated from the infeasible path σ, it assigns potentially one formula to every state of the path. Then its formulas are mapped to the precision. The interpolant gives us information about why the counterexample is unfeasible, therefore using its formulas can help us to eliminate the problem of finding the same spurious counterexample. It can also map *true* of *false* to a state, meaning that it do not have to be refined. When using predicates (in predicate or product abstraction), the interpolant formulas can be used as new predicates. In case of explicitly tracked variables (in explicit-value analysis and product abstraction), the variables of these formulas can be extracted and added to the explicitly tracked variables set. Then the new precisions are simply joined to the old ones.

The final step of refinement is pruning the ARG. This cuts back the the ARG until the earliest state where the refinement occurred. This way the abstraction does not have to start again from scratch, but it can continue constructing the ARG from a place where an actual change occurred. Therefore the algorithm has to find the node (l_i, s_i) with the lowest index for which $I_i \notin \{true, false\}$. Then pruning happens by removing the subtree rooted in node (l_i, s_i) with all of its successors and covered-by edges. Then the abstraction acan continue to construct the ARG from this location.

2.2.3 CEGAR loop

Algorithm 3 presents the behavior of the CEGAR loop. It shows how are abstraction and refinement connected. The inputs of the algorithm are the initial location l_0, the error location l_E, an abstract domain D_L with locations, an initial precision π_0 and a transfer function T_L. The first step is to initialize an abstract reachability graph with only a node corresponding to the initial location l_0 and a $\top$ element. Then, the initial precision π_0 is set as the current precision π. After that the algorithm iterates between abstraction and refinement until abstraction concluded that the error location l_E is not reachable int the ARG or refinement founds a real counterexample.

Algorithm 3: CEGAR loop. [23]

```
   Input : $l_0$: initial location
           $l_E$: error location
           $D_L = (S_L, \bot_L, \sqsubseteq_L, \mathsf{expr}_L)$: abstract domain with locations
           $\pi_0$: initial precision
           $T_L$: transfer function with locations
   Output: safe or unsafe
 1 $ARG := (N := (l_0, \top), E := \emptyset, C := \emptyset)$
 2 $\pi := \pi_0$
 3 while $true$ do
 4     result, $ARG :=$ ABSTRACTION($ARG, l_E, D_L, \pi, T_L$)
 5     if result = $safe$ then return $safe$;
 6     else
 7         result, $\pi_L, ARG :=$ REFINEMENT($ARG, l_E, \pi$)
 8         if result = $unsafe$ then return $unsafe$;
 9     end
10 end
```

Chapter 3

Product abstraction strategies

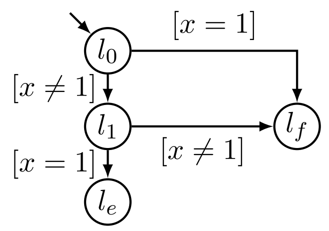
This chapter presents our approaches for combining explicit-value analysis and predicate abstraction into various efficient product abstraction strategies. When combining the two different abstractions, a very important question arises: What should be the new precision when refining the abstraction? 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 some variable but adding another one to the explicitly tracked set.

Our first idea is to always start by adding a new variable to the explicitly tracked variable set. The motivation behind this was that knowing the exact value of a variable yields more information than a predicate and handling predicates is also more expensive computationally. However the downside of explicit-value analysis is that the abstract state space might start growing exponentially if a variable has a large number of different values and some problems might even not be decidable due to unknown ($\top$) values.

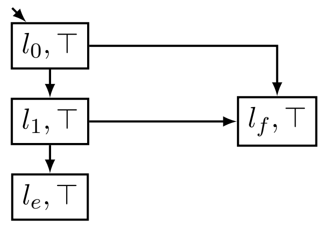
Consider the example CFA in Figure 3.1a, which first checks if $x \neq 1$ and then if $x = 1$ (which obviously cannot be possible). Suppose, that no variables are tracked initially. In this case, the error location is trivially reachable and the refiner extends the precision by adding x. In Figure 3.1b, 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 problem can be that instead of assigning $\top$ to x at l_1, we start to enumerate all the possible values x can take. 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 a more effective solution than using a $\top$ value or using predicates. However, in this case this strategy 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. Before expanding the successors of an abstract state during the abstraction phase, we evaluate every expression on the outgoing edges of the current state. When an expression cannot be evaluated deterministically, we start to enumerate the possible values



(a) Simple CFA.



(b) Corresponding ARG with $\pi_e : \{x\}$.

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

of each tracked variable. We count the number of the different values of a variable and if its greater than k, we stop enumerating, add the variable to a special *dropouts* set, and discard the variable from the precision's explicitly tracked variable set. Then we expand the current state with the transfer function and the newly adjusted precision, which will now not lead to state space explosion, since the variables that can have a large number of values have been eliminated.

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 due to the limit, 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 against the limit k. Examining the possibilities yielded three different strategies [1]. These limit the number of values of an explicitly traced variable based on a single state, or a path, or the whole ARG. The path- and ARG-based strategies also work if we use $\top$ values when an expression cannot be evaluated., since the limit can still be reached on the whole path or in the ARG. This means that we do not check the number of different values in the reachable states, only in the states that have been previously examined. That way, when we start to expand the state with the transfer function, $\top$ will be assigned to every variable that can have more one different values. In case of the state-based strategy, this method is useless, because using $\top$ values instead of enumerating would only result in 0 or 1 different value, therefore we would never reach the limit. That is why we implemented two different algorithms for the path- and ARG-based strategies, one which enumerates all possible states, and one which uses $\top$ values instead.

In addition to these strategies, this thesis presents another idea of combining explicit-value analysis and predicate abstraction. In the previously described strategies, there can be both explicitly tracked variables and predicates present in the precision. However this might not always be the right approach, since handling both kinds of precision at the same time could complicate the algorithm too much while using either only explicit-value analysis or predicate abstraction could solve the problem easily. Therefore we also wanted to create strategies where the precision can contain either only explicitly tracked variables or predicates. However an important question also arises here : Which domain should we use and when? To answer this question, we created two additional algorithms, which will be discussed in detail in Section 3.3.

The goal of this thesis is not only the development of new algorithms but also formulating them using the abstraction and refinement algorithms presented earlier. To be able to seamlessly integrate these 7 strategies in total into the general CEGAR framework discussed in Section 2.2 , we also had to make some important modifications to the framework.

3.1 Modification of the common CEGAR framework

The common CEGAR algorithm is discussed in Section 2.2. The core of the proposed algorithms is the enumeration and limitation of explicit values. This should happen when checking a node in the abstraction phase. Therefore we extended the abstraction algorithm with an additional *precision adjustment* step. This precision adjustment step can be different for the different strategies. In addition, if the step does not perform any action (just leaves the precision as is), we get back the original algorithm. The precision adjusting method is called after checking if a node is unsafe or can be covered, and before it is expanded, so the expansion can be performed without state space explosion. The modified abstraction algorithm is shown in line 14 of Algorithm 4.

We also had to define the new precision adjusting algorithm for the refiner (this step can be found at the 6th line of Algorithm 2), to be able to select between predicates and explicit values. The new Precision adjusting algorithm is presented by Algorithm ....

When giving an overview of the newly proposed algorithms in the previous section, a special *dropouts* set was also mentioned. This set contains the variables which have been explicitly tracked but the number of their different values reached the given limit, and now we want to remove them from the explicitly tracked variable set, and track predicates containing these variables instead. This means that this set must be available and known in the abstraction phase, so that these variables can be immediately removed from the precision before expanding the successors of a node. At the same time it has to be available in the refining phase too, to know when to add predicates instead of explicit values to the precision. This motivated our decision to make the dropouts set a part of the product precision, since the precision is also available from both phases. Therefore from now on the presicion is as follows: $\pi = (\pi_p, \pi_e, \mathsf{dropouts})$. It contains a set of tracked predicates and two variable set: an explicitly tracked variable set and the dropouts set. This modification does not affect the CEGAR algorithms that does not require a dropouts set, since this set can easily be ignored.

To summarize the main changes in the framework, we modified the abstractor by adding a precision adjustment phase to it so now it is also able to modify the precision. We added a *dropouts* set to the precision, so it is reachable both from the abstractor and refiner. And last but not least, we implemented the product abstraction strategies as precision adjustment algorithms in the abstractor and refiner. The next section elaborates these algorithms in more detail.

We also have ideas to further improve the performance of the newly created product algorithms.

Information sharing between domains. One idea is that it could be beneficial if the two different sets of precisions would be able to share information with one and other.

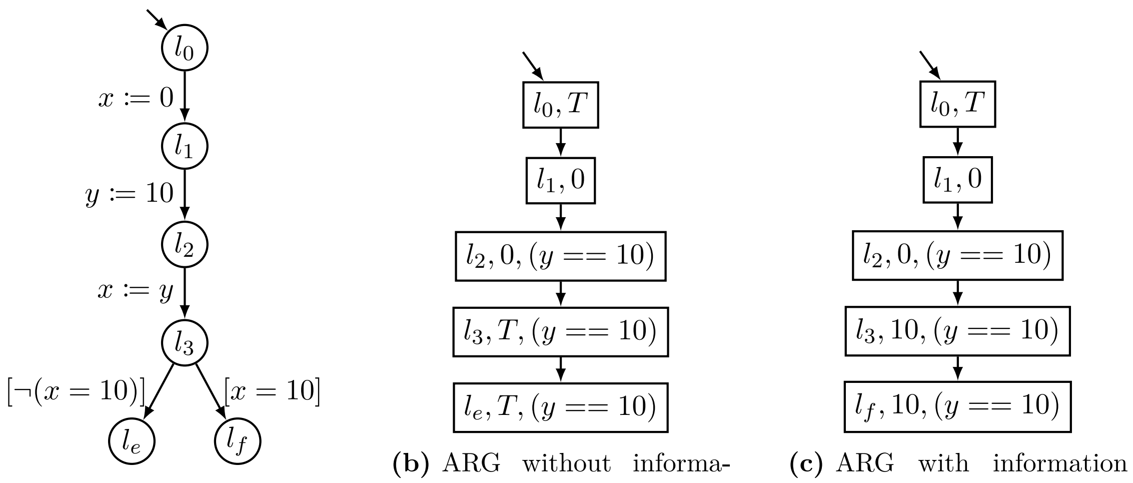
Algorithm 4: Abstraction algorithm.

```
Input  : $ARG = (N, E, C)$: partially constructed abstract reachability graph
         $l_E$: error location
         $D_L = (S_L, \bot_L, \sqsubseteq_L, \mathsf{expr}_L)$: abstract domain with locations
         $\pi_L$: current precision
         $T_L$: transfer function with locations
Output: (safe or unsafe, $ARG$)
1  waitlist $:=$ unmarked nodes from $N$
2  while waitlist $\neq \emptyset$ do
3      $l, s :=$ remove from waitlist
4      // Check if $(l, s)$ is unsafe
5      if $l = l_E$ then
6          return (unsafe, $ARG$)
7      end
8      // Check if $(l, s)$ can be covered
9      else if $\exists (l', s') \in N : (l, s) \sqsubseteq_L (l', s')$ then
10         $C := C \cup \{(l, s, l', s')\}$ // Add covered-by edge
11     end
12     // Otherwise $\pi_L$ get adjusted and $(l, s)$ gets expanded
13     else
14         $\pi_L :=$ PrecisionAdjustment$(\pi_L, (l, s))$
15         foreach $(l', s') \in T_L((l, s), \pi_L) \setminus \bot_L$ do
16             waitlist $:=$ waitlist $\cup \{(l', s')\}$
17             $N := N \cup \{(l', s')\}$ // Add new node
18             $E := E \cup \{(l, s, op, l', s')\}$ // Add successor edge
19         end
20     end
21 end
22 return (safe, $ARG$)
```

This means that the set of predicates and the explicitly tracked variables set are visible for each other when enumerating new successor states.

Consider the example CFA in Figure 3.2a. It first assigns 0 to its variable x then 10 to its variable y. In the next step, y is assigned to x and at the end it is checked whether x equals to x. This CFA is obviously safe but Figure 3.2b shows what happens we decide to track x explicitly with the $(y == 10)$ predicate. In the first two steps both variables are assigned and therefore we reach $(l_2, 0, (y == 10))$ with the value of x being 0, and $(y == 10)$ being true. However when y is assigned to x in the next step, the value of x becomes $\top$, since the predicates are not visible for the explicitly tracked values. From the standpoint of explicit value analysis y is not tracked, therefore its value is unknown. At the end, with the value of x being $\top$, the error location is reachable and a spurious counterexample is found. However Figure 3.2c depicts the ARG created with the same precision, but this time the two precision sets are visible to each other. This way when assigning y to x, because it is known from the predicate that the value of y is 10, x is also assigned 10. At the end, the CFA can be successfully verified with information sharing.

Therefore now a special flag can be used to turn on or off information sharing between the two domains.



(a) Simple CFA. **(b)** ARG without information sharing. **(c)** ARG with information sharing.

Figure 3.2: Simple CFA and its corresponding ARG $\pi_e : \{x\}$, $\pi_p : \{(y == 10)\}$, with sharing information turned on and off.

Changing the order of coverage check. Another idea is that changing the order in which coverage is checked during abstraction can further improve the speed of the algorithms. Since a product state contains a state for predicate abstraction and another for explicit value analysis, both of them have to be covered by the corresponding states of another product state, to be able to add a covering edge.

Originally, coverage was checked by first examining the coverage of the product abstraction states, then if it could be covered, the explicit value analysis states are also checked. However since handling predicates is more expensive computationally, maybe examining the explicit states first could produce faster results. Therefore now it is an option to choose which order we would like to check the coverage.

We also added a third option, to check both kinds of states at the same time.

3.2 Strategies tracking explicit values and predicates simultaneously

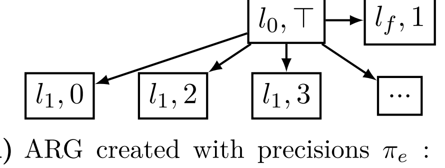
3.2.1 Limit number of successors based on a single state

In the first strategy (Algorithm 5), we count the different values of the tracked variables when enumerating successors for a given state. We start by setting a restart flag which will be used later. Then we start enumerating the reachable states based on the operations on the outgoing edges of the source state and we examine the values of the explicitly tracked variables. If the number of the different values of a variable in the reachable states exceeds k, we add it to the precision's *dropouts* set. We also remove it from the precision π_e explicitly tracked variable set, 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 reachable states to list and no variable was removed, we do not need to restart and we can return the modified precision.

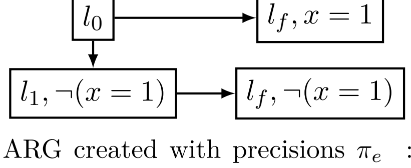
Example 5. *Consider the example CFA in Figure 3.1a again. If we use the state-based product abstraction, the variable x is added to the set of explicitly tracked variables as pre-*

```
Algorithm 5: State-based precision adjustment
Input  : $s$: source state
         $(\pi_p, \pi_e \mathsf{dropouts})$: current precision
         $k$: bound for values of explicitly tracked variables
Output: $(\pi_p, \pi_e, \mathsf{dropouts})$: refined precision
1  $Ops_s$ := operations from the outgoing edges of $s$
2  foreach $op \in Ops_s$ do
3      restart := false
4      while restart do
5          assignments := map the different assignments for each variable in $\pi$ for the
            formula $s \wedge op$
6          if more than $k$ different values are possible for a variable $v_i$ then
7              dropouts := dropouts $\cup\, v_i$
8              $\pi_e := \pi_e \setminus v_i$
9              restart := true
10         end
11     end
12 end
13 return $(\pi_p, \pi_e \mathsf{dropouts})$
```

viously ($\pi_e = \{x\}$). The corresponding ARG for this precision can be seen in Figure 3.3a. The program starts at state $(l_0, \top)$, 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 3.3b shows the ARG created with the new 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 precisions $\pi_e : \{x\}$ and $\pi_p : \{\}$.



(b) ARG created with precisions $\pi_e : \{\}$ and $\pi_p : \{(x = 1)\}$.

Figure 3.3: ARGs created with the state-based strategy.

3.2.2 Limit number of values on a path

The previous strategy only counted different values for the reachable states 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 before we expand it. Algorithm 6 presents the procedure for this strategy, which is similar to the state-based. The main difference is that before we start to examine the reachable states, we collect the ancestors of the current state and map the different values occurring in them for each explicitly tracked variable. After this step, the algorithm is the same as the state-based one: We start to enumerate the reachable states of the current state while always updating the map containing the different values for each explicitly tracked variable. If the number of different values of a variable reach k, we add this variable to the *dropouts* set, remove it from the precision and set the restart flag.

Algorithm 6: Path-based precision adjustment

```
Input  : s: source state
         (π_p, π_e dropouts): current precision
         k: bound for values of explicitly tracked variables
Output: (π_p, π_e, dropouts): refined precision
1  assignments := map the different values for each variable in π in the ancestor
    states
2  Ops_s := operations from the outgoing edges of s
3  foreach op ∈ Ops_s do
4      restart := false
5      while restart do
6          assignments := map the different assignments for each variable in π for the
            formula s ∧ op
7          if more than k different values are possible for a variable v_i then
8              dropouts := dropouts ∪ v_i
9              π_e := π_e \ v_i
10             restart := true
11         end
12     end
13 end
14 return (π_p, π_e dropouts)
```

Example 6. *Consider the example CFA in Figure 3.4. 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 3.5a). 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 3.5b. The program starts at (l_0), where the*

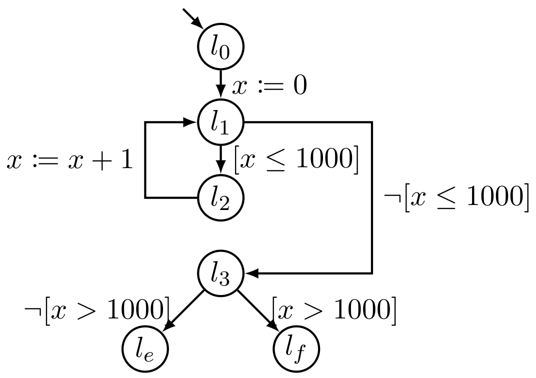
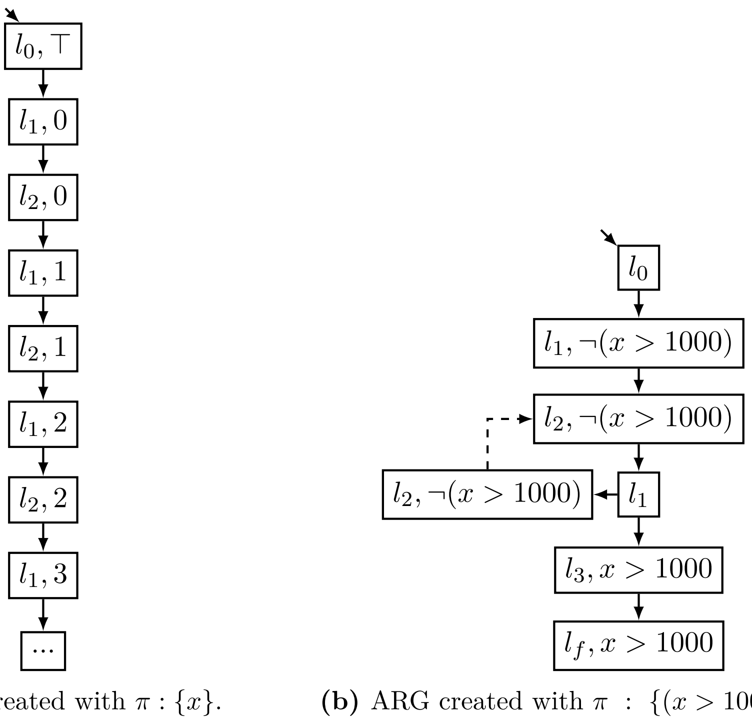


Figure 3.4: Example CFA.

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 3.5: AGRs created with the path-based strategy.

As mentioned previously, we proposed another algorithm for the path-based strategy, which does not enumerate all the possible states when an expression cannot be evaluated, but uses the $\top$ value instead. This way the limit can still be reached on the whole path leading to a state. That is why we check the number of values of the explicitly tracked variables before enumerating the successors of a state. Algorithm 7 presents the procedure.

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. This way when there are multiple values in the successors, we use the $\top$ value in the transfer function instead of enumerating them.

Algorithm 7: Path-based precision adjustment without enumeration

```
Input  : $s$: source state
         $(\pi_p, \pi_e \mathsf{dropouts})$: current precision
         $k$: bound for values of explicitly tracked variables
Output: $(\pi_p, \pi_e, \mathsf{dropouts})$: refined precision
1 assignments := map the different values for each variable in $\pi$ in the ancestor
    states
2 if more than $k$ different values are possible for a variable $v_i$ then
3     dropouts := dropouts $\cup$ $v_i$
4     $\pi_e := \pi_e \setminus v_i$
5 end
6 return $(\pi_p, \pi_e \mathsf{dropouts})$
```

3.2.3 Limit number of values in ARG

Our third strategy examines the number of the different values in the whole ARG. It examines the variables in all of the previous states through the whole ARG. Algorithm 8 presents the procedure, similar to the previous strategies, 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. Therefore the map containing the different values for each variable is not recreated each time the method is called, but stored in it and updated every time a new stated is examined.

For the ARG-based strategy, we also have another algorithm where we use $\top$ values instead of enumerating all possible states, which is presented in Algorithm 9. It works by the same principle as the path-based non-enumerating algorithm. If a variable's number of values exceeded the limit k, we add this variable to the *dropouts* set, remove it from the precision and then we g on to use the transfer function using $\top$ values to calculate the successors.

3.3 Strategies switching between explicit values and predicates

The theory of these strategies are very similar to the ARG-based strategy. The main principle is that we start with explicit value analysis, we introduce a k limit, count the different values in the ARG before expanding a state and adjust the precision, just like in the ARG-based. The new strategies also use the same refiner. The difference is that after the limit has been reached, the new algorithms clear the set of explicitly tracked variables, add every variable of the program to the special *dropouts* set and therefore switching solely to predicate abstraction.

What distinguishes the two new algorithms is that in one algorithm k limits the number of different values of a single variable, while in the other it limits the number of different values of every variable of the explicitly tracked variable set.

Algorithm 8: ARG-based precision adjustment

```
Input   : $s$: source state
          $(\pi_p, \pi_e \mathsf{dropouts})$: current precision
          $k$: bound for values of explicitly tracked variables
Output: $(\pi_p, \pi_e, \mathsf{dropouts})$: refined precision
1  $\mathsf{assignments} :=$ map the different values for each variable in $\pi$ in the ARG
2  $Ops_\mathsf{s} :=$ operations from the outgoing edges of $s$
3  foreach $op \in Ops_s$ do
4      $\mathsf{restart} :=$ false
5      while restart do
6          $\mathsf{assignments} :=$ map the different assignments for each variable in $\pi$ for the
            formula $s \land op$
7          if more than $k$ different values are possible for a variable $v_i$ then
8              $\mathsf{dropouts} := \mathsf{dropouts} \cup v_i$
9              $\pi_e := \pi_e \setminus v_i$
10             $\mathsf{restart} :=$ true
11         end
12     end
13 end
14 return $(\pi_p, \pi_e \mathsf{dropouts})$
```

Algorithm 9: ARG-based precision adjustment without enumeration

```
Input   : $s$: source state
          $(\pi_p, \pi_e \mathsf{dropouts})$: current precision
          $k$: bound for values of explicitly tracked variables
Output: $(\pi_p, \pi_e, \mathsf{dropouts})$: refined precision
1  $\mathsf{assignments} :=$ map the different values for each variable in $\pi$ in the ARG
2  if more than $k$ different values are possible for a variable $v_i$ then
3      $\mathsf{dropouts} := \mathsf{dropouts} \cup v_i$
4      $\pi_e := \pi_e \setminus v_i$
5  end
6  return $(\pi_p, \pi_e \mathsf{dropouts})$
```

3.4 Related work

The combination of different abstract domains have been studied in the literature before. To be able to thoroughly evaluate our results achieved with our newly introduced strategies, we also wanted to compare them to other already existing, similar algorithms. Therefore, we also implemented two strategies in the THETA framework.

3.4.1 Dynamic precision adjustment

The dynamic precision adjustment approach [9] for the explicit and predicate domains is similar to our ARG-based strategy. The main difference is that while both this and our approaches are able to adjust the precision in the abstraction phase, our algorithms are only able to reduce the size of the explicitly tracked variable set, while dynamic precision adjustment also adds new predicates during abstraction. In other words, dynamic precision adjustment behaves like we combined the precision adjusting algorithms of the abstractor

Algorithm 10: Domain switching precision adjustment based on a single variable

```
Input  : $s$: source state
         $(\pi_p, \pi_e \mathsf{dropouts})$: current precision
         $k$: bound for values of explicitly tracked variables
Output: $(\pi_p, \pi_e, \mathsf{dropouts})$: refined precision
1  $\mathsf{assignments} :=$ map the different values for each variable in $\pi$ in the ARG
2  $Ops_{\mathsf{s}} :=$ operations from the outgoing edges of $s$
3  foreach $op \in Ops_s$ do
4      $\mathsf{restart} :=$ false
5      while restart do
6          $\mathsf{assignments} :=$ map the different assignments for each variable in $\pi$ for the
            formula $s \wedge op$
7          if more than $k$ different values are possible for a variable $v_i$ then
8              $\mathsf{dropouts} :=$ every variable of the program
9              $\pi_e := \emptyset$
10         end
11     end
12 end
13 return $(\pi_p, \pi_e \mathsf{dropouts})$
```

and the refiner. This also means that this strategy does not adjust the precision during refinement.

Algorithm 12 presents the dynamic precision adjustment strategy we implemented in Theta. It starts out the same way as the ARG-based strategy: It examines the variables in all of the previous states through the whole ARG and also in the reachable states from the current one. But when the number of different values for a variable reaches limit k, we not only remove this variable from the set of explicitly tracked variables and add it to the *dropouts* set, but also add a predicate containing this variable to the predicate set. To obtain an appropriate predicate, we examine the statements of the edges of the CFA. If an assumption statement containing the variable is found, it is added to the precision as a predicate.

3.4.2 Changing domain based on counterexample

Counterexample-based product abstraction [10] focuses on the counterexamples returned by the abstractor. This strategy is different form every previously discussed one, because it does not modify the precision during abstraction. It does not keep explicitly tracked variables and predicates at the same time in the precision, but rather switches domains when the same counterexample is found multiple times. Therefore for this algorithm, a new refiner was created.

Algorithm **??** presents the counterexample based strategy. It also starts with explicit value analysis, but beside adjusting the precision, the refiner also examines and stores the received counterexamples. If a counterexample is already present in the cache, the algorithm changes from explicit value analysis to predicate abstraction.

Algorithm 11: Domain switching precision adjustment based on every variable

```
Input  : $s$: source state
         $(\pi_p, \pi_e$dropouts$)$: current precision
         $k$: bound for values of explicitly tracked variables
Output: $(\pi_p, \pi_e,$ dropouts$)$: refined precision
1  assignments := map the different values for each variable in $\pi$ in the ARG
2  $Ops_s$ := operations from the outgoing edges of $s$
3  foreach $op \in Ops_s$ do
4      restart := false
5      while restart do
6          assignments := map the different assignments for each variable in $\pi$ for the
            formula $s \wedge op$
7          if more than $k$ different values are possible in total $v_i$ then
8              dropouts := every variable of the program
9              $\pi_e := \emptyset$
10         end
11     end
12 end
13 return $(\pi_p, \pi_e$dropouts$)$
```

Algorithm 12: Dynamic precision adjustment

```
Input  : $s$: source state
         $(\pi_p, \pi_e$dropouts$)$: current precision
         $k$: bound for values of explicitly tracked variables
Output: $(\pi_p, \pi_e,$ dropouts$)$: refined precision
1  assignments := map the different values for each variable in $\pi$ in the ARG
2  $Ops_s$ := operations from the outgoing edges of $s$
3  foreach $op \in Ops_s$ do
4      restart := false
5      while restart do
6          assignments := map the different assignments for each variable in $\pi$ for the
            formula $s \wedge op$
7          if more than $k$ different values are possible for a variable $v_i$ then
8              dropouts := dropouts $\cup\, v_i$
9              $\pi_e := \pi_e \setminus v_i$
10             foreach edge in CFA do
11                 if edge has an assumption stmt containing $v_i$ then
12                     $\pi_p := \pi_p \cup stmt$
13                     break
14                 end
15             end
16             restart := true
17         end
18     end
19 end
20 return $(\pi_p, \pi_e$dropouts$)$
```

Chapter 4

Evaluation

This chapter presents our implementation of the nine product abstraction-based strategies and the evaluations of these algorithms including a comparison to explicit-value analysis and predicate abstraction. We proposed different research questions and run measurements so each of these questions can be answered.

4.1 Implementation

A goal of this thesis is to implement the precision adjuster algorithms and the modifications discussed in Chapter 3 in the open source[1] Theta framework [30, 24], 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 [25, 30]. Furthermore, Theta uses Z3 [18] as an SMT solver.

We also had to modify the runnable tool which is deployed in a jar file named `theta-cfa-cli.jar` to be able to run the algorithms with command line arguments. The used 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 abstract domain to run. The used values are `EXPL` for explicit-value analysis, `PRED` for predicate abstraction and `PROD_PRED_EXPL` for the product abstraction of predicate abstraction and explicit-value analysis. It is also a mandatory argument.
- `refinement`: This is the refinement strategy. In case of the counterexample-based strategy the used value is `CEX_BASED`, for every other it is `SEQ_ITP` (this refers to the sequence interpolation [24]).
- `precadjust`: With this flag the precision adjustment strategy of the abstractor can be defined. The possible values are:
  - `STATE` for the state-based (Section 3.2.1),
  - `PATH_NO_T` for the path-based (Section 3.2.2),
  - `PATH_T` for the path-based using $\top$ values (Section 3.2.2),

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

  - `ARG_NO_T` for the ARG-based (Section 3.2.3),
  - `ARG_T` for the ARG-based using $\top$ values (Section 3.2.3),
  - `ARG_ONE_PRED` for the domain switching based on a single variable (Section 3.3),
  - `ARG_WHOLE_PRED` for the domain switching based on all variables (Section 3.3),
  - `DYNAMIC` for the dynamic precision adjuster strategy (Section 3.4.1)
  - and `NO_OP` which is used if no precision adjustment is needed during abstraction (for predicate abstraction, explicit-value analysis and the counterexample-based strategy in Section 3.4.2).

- `limit`: This is the limit k for product abstraction. It is an optional parameter with a default value of 5.
- `share`: With this flag information sharing in product abstraction can be turned on or off.
- `secondFirst`: With this flag, the order of coverage checking can be set. The possible values are 0 if we want to leave the order unchanged (in case of our strategies this mean that predicates are examined first), 1 if we want to change the order or 2 if we want to check the coverage of both of them at the same time.

4.2 Measurement configuration

We ran the measurements on a 64 bit Ubuntu 18.04 operating system, with the tool RunExec fron the BenchExec suite [11]. 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) [4].

We evaluated 548 input programs from five different sources and categories: plc, eca, locks, loops and ssh. The 90 programs in plc are industrial programmable logic controller (PLC) codes from CERN [21], while the other four categories come from the Competition on Software Verification (SV-COMP) [4, 5]. 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 156 programs with small locking mechanisms described with non-deterministic integers and if-then-else constructs. The 105 programs in category loops describe tasks focusing on loops. The programs in category ssh describe 17 large server-client systems.

We evaluated these programs based on the following research questions:

RQ1 How do the basic state-, path- and ARG-based strategies perform compared to each other?

RQ2 How do the basic path- and ARG-based strategies perform compared to their counterparts using $\top$ values?

RQ3 How do the strategies tracking predicates and explicit values at the same time perform compared to the domain switching strategies?

RQ4 How do the different product abstraction strategies perform for different limits?

RQ5 How do the different product abstraction strategies perform with information sharing between the two domains?

RQ6 How do the different product abstraction strategies perform with different coverage checking orders?

RQ7 How do our product abstraction strategies perform compared to related work?

RQ8 How do the product abstraction strategies perform compared to predicate abstraction and explicit-value analysis?

We ran measurements for every question on every model, but not with every possible configuration. This decision was made because we would have more than 88 different configurations, which proved to be too much to handle. Therefore we used the best performing result from the previous measurements for the comparison everywhere it was possible. For example in the case of *RQ8* we chose to present only the best performing product abstraction strategy to compare with predicate abstraction and explicit-value analysis. Also *RQ4* has not been evaluated individually, we ran measurements with different limits for every other question. We enforced a time limit of 500 seconds and a memory limit of 4 GB. 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 evaluate the results of the measurements for every proposed question.

4.3 Results

This section presents the results of the measurement ran to evaluate the proposed research questions in the previous section.

4.3.1 RQ1: Basic state-, path- and ARG-based strategies

To evaluate *RQ1*, we ran measurements with six different configurations: the state-based, and the path- and ARG-based strategies without using $\top$ values. The limits were set at 1 and 32. The other values were left at default, meaning that the information sharing between the two domains was turned off, and the coverage checking started with predicate abstraction states.

Precision adjustment	Limit	Succ. count	Total time (ms)
ARG_NO_T	1	444	18358
PATH_NO_T	1	443	18467
STATE	1	437	11457
ARG_NO_T	32	415	7753
PATH_NO_T	32	415	8603
STATE	32	395	7390

Table 4.1: Comparison of the state-, path and ARG-based strategies

Table 4.1 show the results of this measurement. The different configurations are ranked form best to worst. The first column shows the precision adjustment algorithm, the second represents the number of the limit, the third shows the number of successfully verified models and in the fourth column the total run time can be seen in milliseconds. The table shows that the best result with 444 successfully verified models was achieved by the ARG-based strategy with limit 1. Only one model behind is the path-based strategy also with limit 1. Based on this result, we can see that with a larger limit, the performance gets worse.

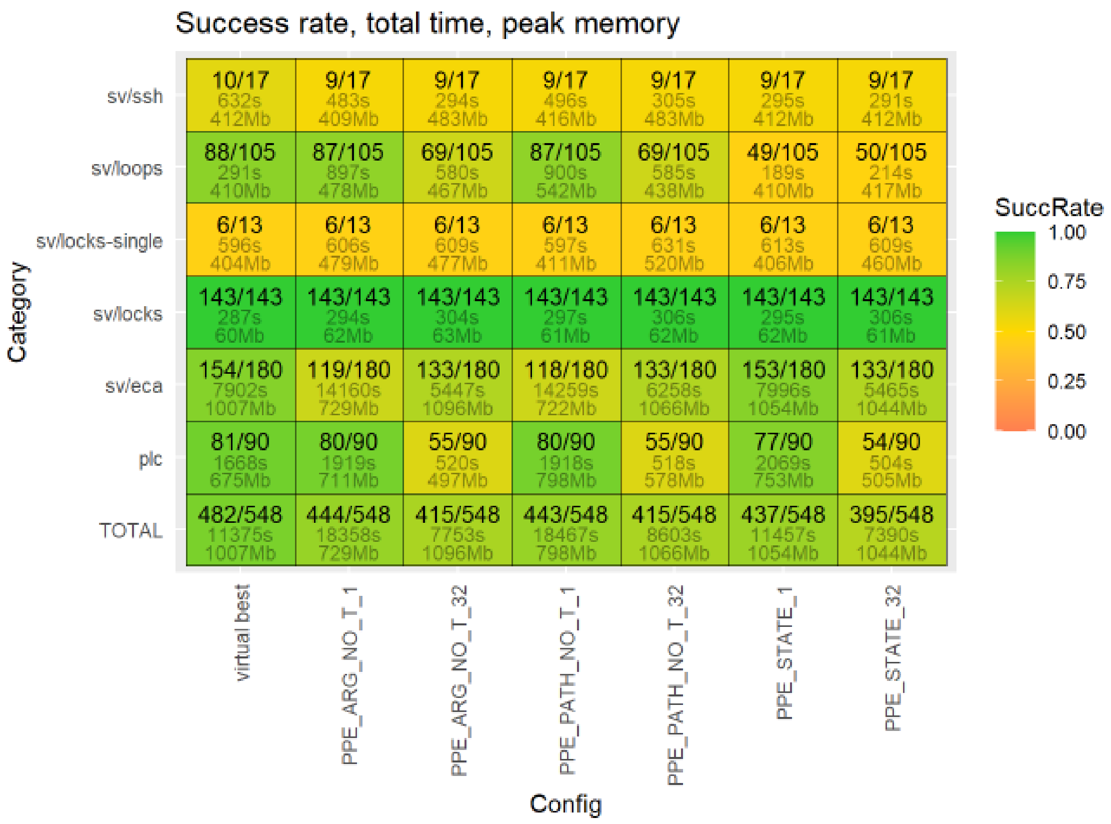


Figure 4.1: Heatmap of the state-, path- and ARG-based strategies

In Figure 4.1 a heatmap can be seen representing the success rate and total runtime of the different configurations in every program category. The greener the tile is, the better the performance. Column virtual best represents the best configuration chosen for every modell. As the heatmap shows, the performance of the different configurations are pretty similar in every category. Generally the configurations with limit 1 have better results with plcs and loops, while they have a slightly worse performance in program category eca. It is also worth noting that every configuration was able to successfully verify all the programs in locks and while the state-based algorithm had a worse overall performance (mainly because of the loops category), with limit 1 it was able to verify more programs than any other configurations in category eca.

4.3.2 RQ2: Path- and ARG-based strategies with and without ⊤

This section presents the results of the evaluation of *RQ2*. We compared the path- and ARG-based strategies that use ⊤ values to the ones that do not. Measurements for the algorithms using top values were run with limits 1 and 32, while from the other category only limit 1 was selected, because it had the better performance in the previous measurements. There other configuration parameters were left unchanged again.

Precision adjustment	Limit	Succ. count	Total time (ms)
ARG_NO_T	1	444	18358
PATH_NO_T	1	443	18467
ARG_T	32	239	11805
PATH_T	32	216	5840
ARG_T	1	175	5523
PATH_T	1	162	6798

Table 4.2: All the measurements ran

Table 4.2 shows the results of this measurement. It is obvious that the algorithms that use $\top$ values have a far worse performance. While they have a generally better performance with a higher limit, they could only verify almost half as much models as the algorithms that do not use $\top$ values.

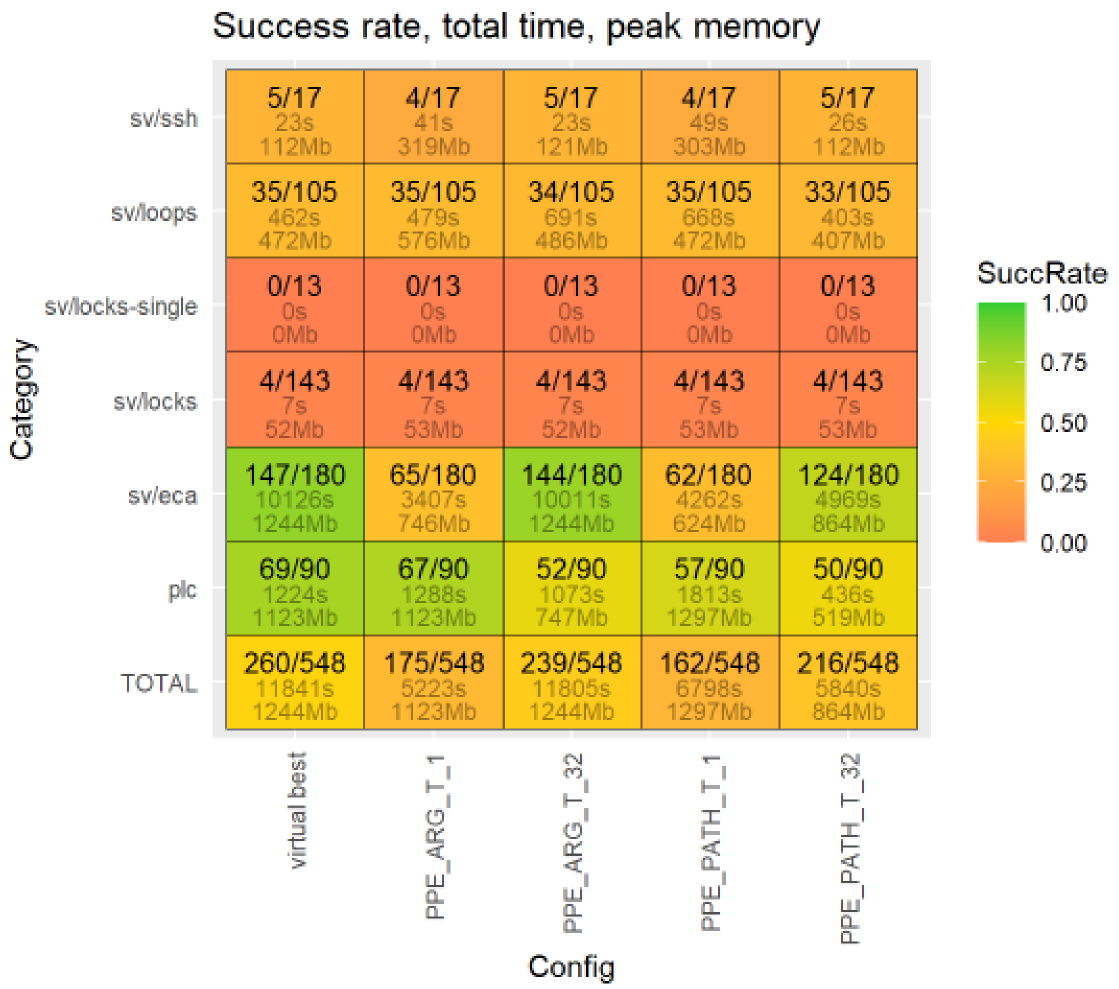


Figure 4.2: Heatmap of the strategies using $\top$

Heatmap 4.2 shows the performance of the configurations for the different program categories. They have generally bad results in every category. The best performance of the $\top$ using algorithms is achieved by the ARG-based one with limit 1, in the category eca, with 144 successfully verified programs of the 180.

In conclusion the algorithms using $\top$ do not measure up to the other base algorithms.

4.3.3 RQ3: Combiner and domain switching strategies

The results of the evaluation of *RQ3* are presented in this section. We compare the best path- and ARG-based strategies again, bu this time with the domain switching algorithms. The domain switching algorithms were also run with limits 1 and 32 and with the other parameters unchanged.

Precision adjustment	Limit	Succ. count	Total time (ms)
ARG_NO_T	1	444	18358
PATH_NO_T	1	443	18467
ARG_ONE_PRED	1	438	14049
ARG_ONE_PRED	32	429	8220
ARG_WHOLE_PRED	1	247	13660
ARG_WHOLE_PRED	32	195	5693

Table 4.3: All the measurements ran

Table 4.3 shows the results of the measurements. While the domain switching algorithm that limits based on one variable is not far behind, the simple path- and ARG-based algorithms still have the best performance. For the domain switching algorithms there is not much difference in the performance between a lower and a higher limit, but the two kinds of algorithms have very different results. The algorithms limiting based on the whole ARG have far worse results.

In Figure 4.3 the heatmap representing the domain switching algorithms can be seen. While the algorithms limiting based on the whole ARG have a worse performance in every category, with limit 1 it almost have as good results as the best configuration in this measurement. The result of the two algorithms limiting based on one variable are not that different, but it is worth noting that while with limit 1 has the best results in category plc and a little worse in eca, the opposite is true for limit 32.

4.3.4 RQ5: Sharing information

In this section, the results of evaluating *RQ5* can be seen. Now we focus on the information sharing between the two different domains. Therefore we chose the best performing configurations form the previous results (which are still the basic path- and ARG-based with limit 1) and ran the measurements now with the information sharing enabled.

Precision adjustment	Limit	Share	Cover order	Succ. count	Total time (ms)
PATH_NO_T	1	true	0	448	13941
ARG_NO_T	1	true	0	447	17353
ARG_NO_T	1	false	0	444	18358
PATH_NO_T	1	false	0	443	18467

Table 4.4: All the measurements ran

The results are presented in Table 4.4 and it shows that we managed to find a slightly better configuration. Although the result are not that different, with the difference being only 5 models between the number of successfully verified models of the best and worst performing configurations of this category.

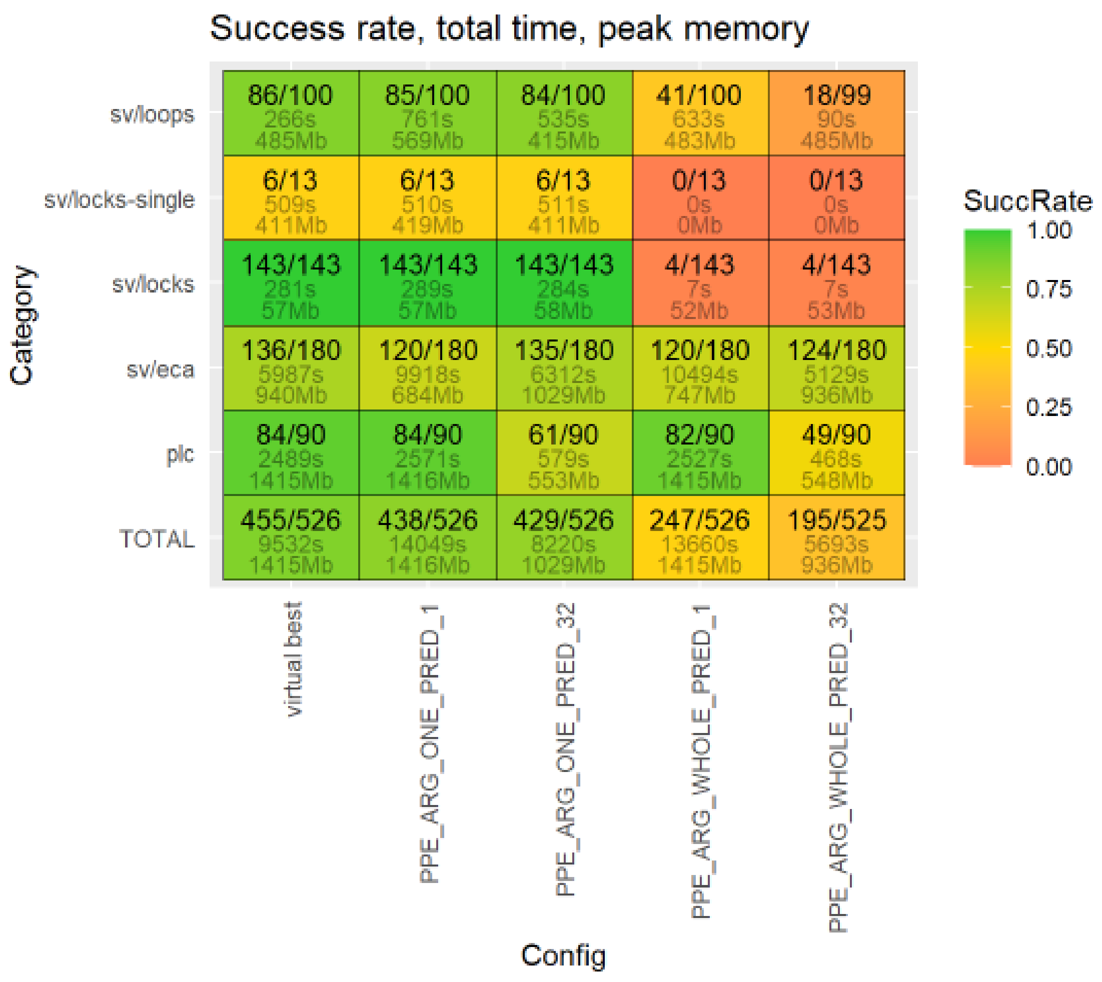


Figure 4.3: Heatmap of the domain switching strategies

The heatmap 4.4 of these configurations are also showing the same results.

4.3.5 RQ6: Coverage checking order

This section focuses on the comparison of the coverage checking order, *RQ6*. For the measurement we chose the best performing configurations from the previous result, which were the path- and ARG based algorithms with limit 1 and information sharing turned on. We ran the measurements with this time changing only the order of the coverage checking.

Precision adjustment	Cover order	Succ. count	Total time (ms)
ARG_NO_T	2	449	13774
PATH_NO_T	2	449	14381
ARG_NO_T	1	448	13226
ARG_NO_T	0	448	13326
PATH_NO_T	0	448	13941
PATH_NO_T	1	447	13708

Table 4.5: All the measurements ran

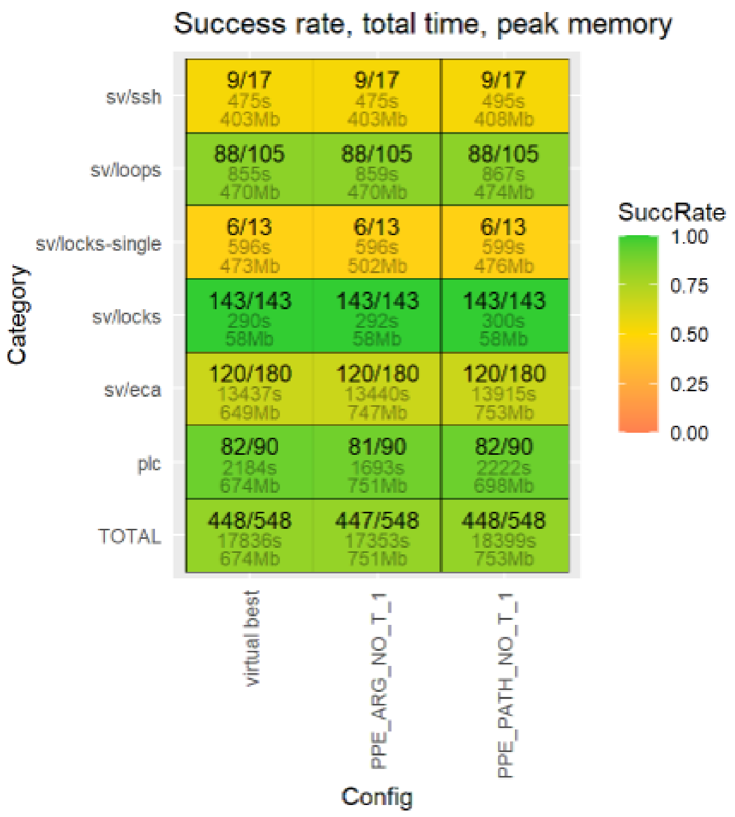


Figure 4.4: Heatmap of the best strategies sharing information

Table 4.5 shows the results of this measurement. It can be seen that changing the coverage checking order between the two domains did not change much, but the configurations checking both of them at the same time managed to produce better results by one model.

Heatmap 4.5 presents the results of the different configurations in the different program categories. This figure also shows that there is no significant change if we change the coverage checking order.

4.3.6 RQ7: Related work

This section presents the results of evaluating *RQ7*: the comparison of our best strategies with related work. The counterexample-based strategy can only have one configuration, but the dynamic precision adjusting algorithm has been run with both limit 1 and 32.

Refinement	Precision adjustment	Limit	Succ. count	Total time (ms)
SEQ_ITP	ARG_NO_T	1	449	13774
SEQ_ITP	PATH_NO_T	1	449	14381
SEQ_ITP	DYNAMIC	32	336	12922
SEQ_ITP	DYNAMIC	1	285	8404
CEX_BASED	NO_OP	-	220	11463

Table 4.6: Comparison of our best configurations and related work

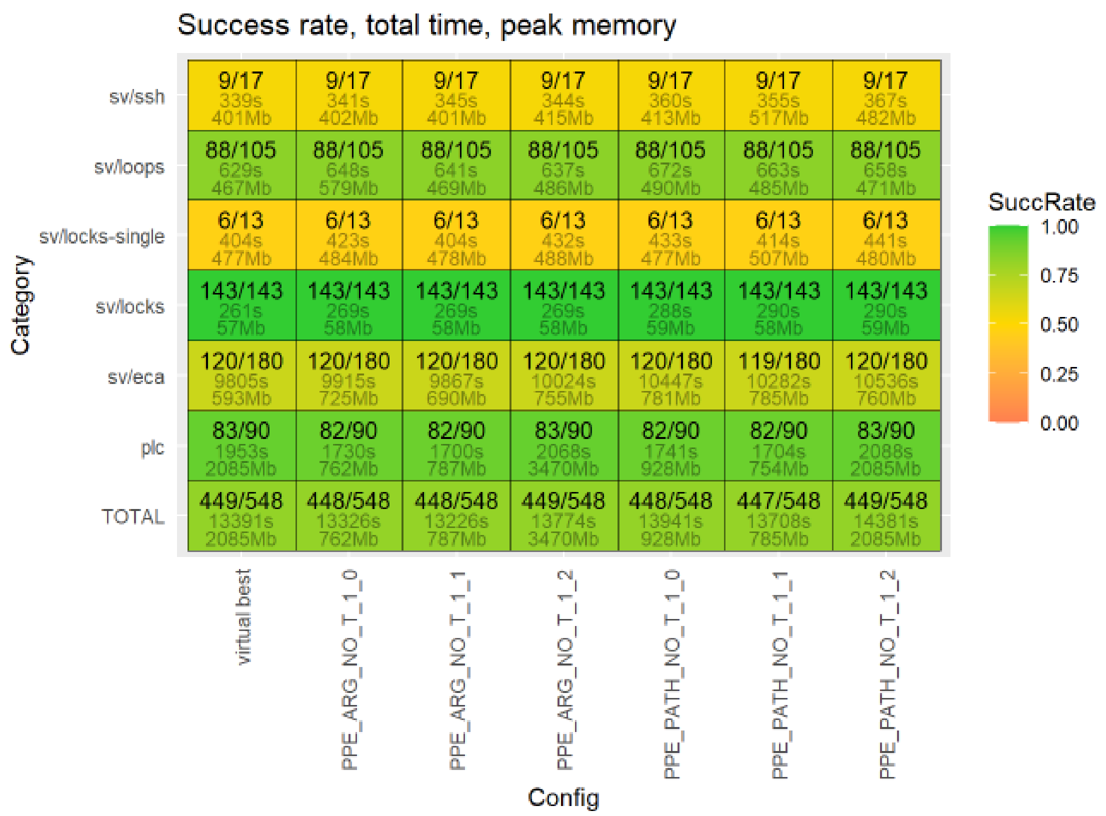


Figure 4.5: Heatmap of the best strategies switching cover orders

Table 4.6 presents the results of the measurement. The counterexample-based strategy did not perform well, it only verified 220 models from the 548 successfully. The dynamic precision adjustment especially with limit 32 had better results with 336 verified models, but it still does not come close to our best results: 449 successfully verified model by both the path- and ARG-based strategies with turned on information sharing, and checking coverage for bot domains at the same time.

Figure 4.6 presents the heatmap showing the performance of the dynamic precision adjustment and counterexample-based algorithms in the different program categories. All of the configurations have bad result in categories locks-single and ssh while having in general good results in eca. The counterexample-based strategy also performs especially bad in category locks. Although it can be observed in this heatmap too, that configurations with higher limits have better results in category eca.

4.3.7 RQ8: Predicate abstraction and explicit-value analysis

This section present the evaluation of our last but maybe most important question: How do the product abstraction based algorithms perform in comparison to the two base algorithms: product abstraction and explicit-value analysis.

Table 4.7 shows the results of the measurement. All in all, though the product abstraction strategies performed better than explicit-value analysis, they still could not measure up to the number of 452 successfully verified models of the predicate abstraction. Although it is worth noting that the difference is very minimal.

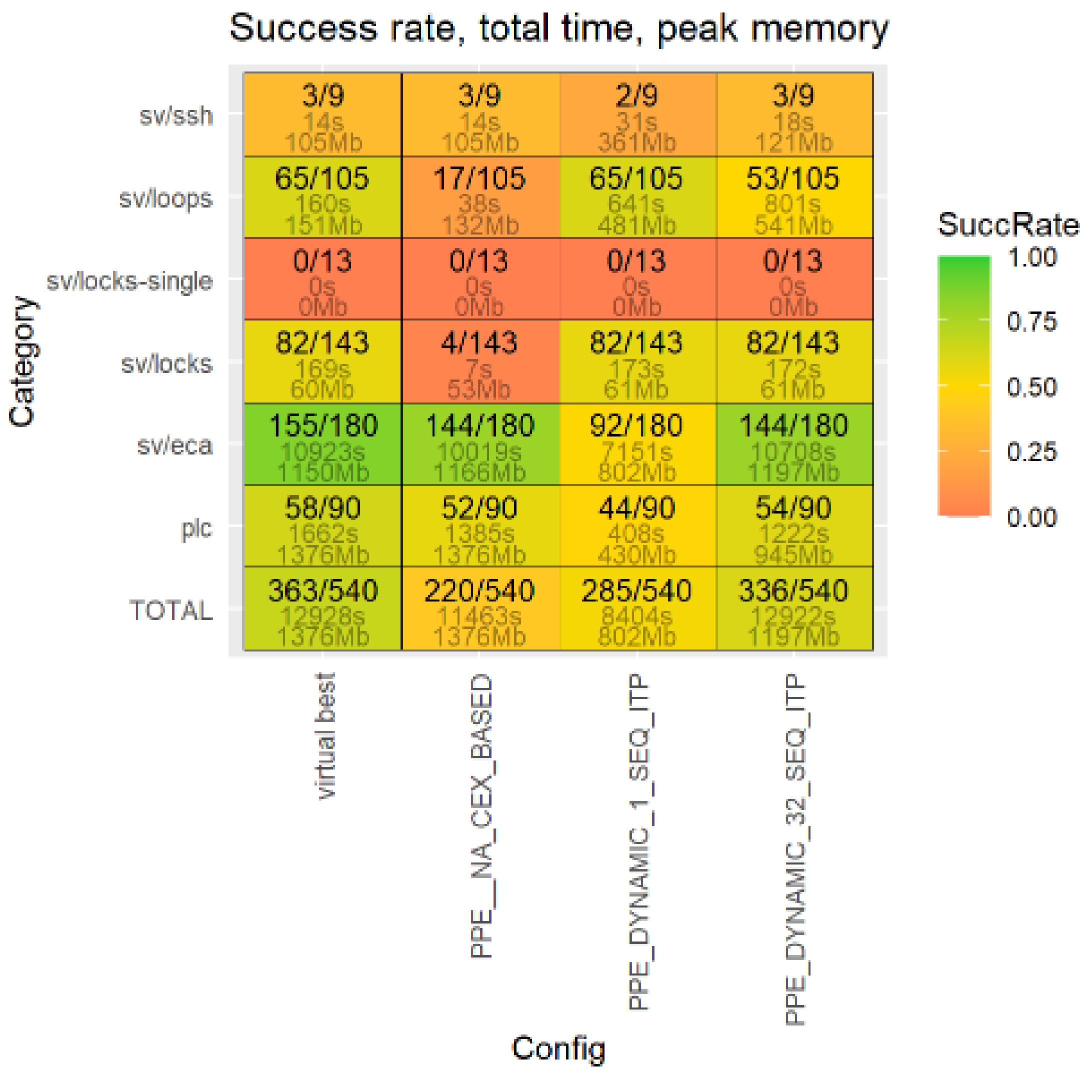


Figure 4.6: Heatmap of the related strategies

Domain	Precision adjustment	Succ. count	Total time (ms)
PRED	NO_OP	452	11110
PROD_PRED_EXPL	ARG_NO_T	449	13774
PROD_PRED_EXPL	PATH_NO_T	449	14381
EXPL	NO_OP	239	8575

Table 4.7: Measurements comparing our best configurations to predicate abstraction and explicit-value analysis

Figure 4.7 shows the heatmap of the predicate abstraction and explicit-value analysis. The explicit-value analysis has very similar results to the performance of the counterexample-based seem in Figure 4.6. Predicate abstraction also has almost the same results as our best performing configurations seen in Figure 4.5.

4.3.8 Summary

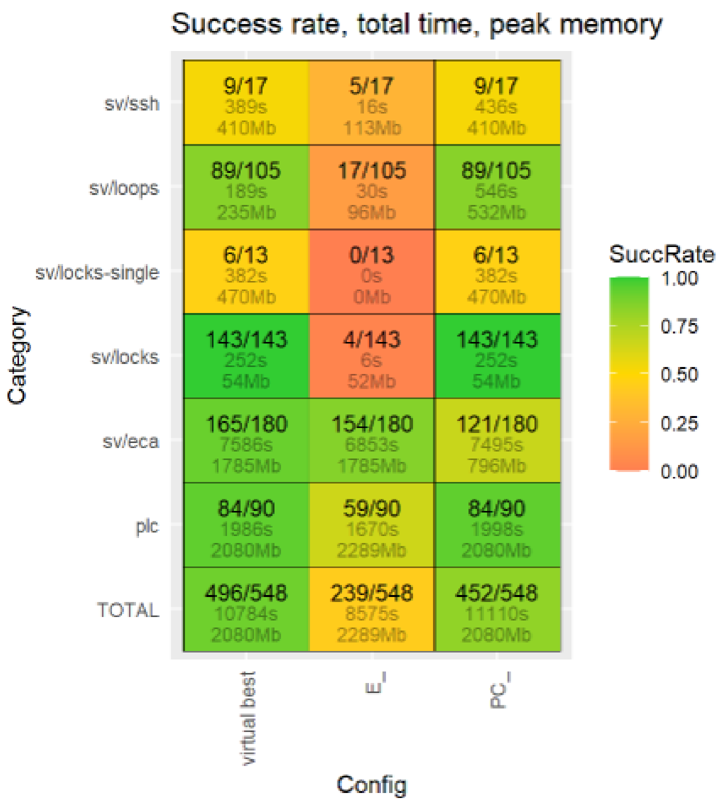


Figure 4.7: Heatmap of predicate abstraction and explicit-value analysis

Domain	Refinement	Precision adjustment	Limit	Share	Cover order	Succ. count	Total time (ms)
PRED	SEQ_ITP	NO_OP	-	-	-	452	11110
PROD_PRED_EXPL	SEQ_ITP	ARG_NO_T	1	true	2	449	13774
PROD_PRED_EXPL	SEQ_ITP	PATH_NO_T	1	true	2	449	14381
PROD_PRED_EXPL	SEQ_ITP	ARG_NO_T	1	true	1	448	13226
PROD_PRED_EXPL	SEQ_ITP	ARG_NO_T	1	true	0	448	13326
PROD_PRED_EXPL	SEQ_ITP	PATH_NO_T	1	true	0	448	13941
PROD_PRED_EXPL	SEQ_ITP	PATH_NO_T	1	true	1	447	13708
PROD_PRED_EXPL	SEQ_ITP	ARG_NO_T	1	true	0	447	17353
PROD_PRED_EXPL	SEQ_ITP	ARG_NO_T	1	false	0	444	18358
PROD_PRED_EXPL	SEQ_ITP	PATH_NO_T	1	false	0	443	18467
PROD_PRED_EXPL	SEQ_ITP	ARG_ONE_PRED	1	false	0	438	14049
PROD_PRED_EXPL	SEQ_ITP	STATE	1	false	0	437	11457
PROD_PRED_EXPL	SEQ_ITP	ARG_ONE_PRED	32	false	0	429	8220
PROD_PRED_EXPL	SEQ_ITP	ARG_NO_T	32	false	0	415	7753
PROD_PRED_EXPL	SEQ_ITP	PATH_NO_T	32	false	0	415	8603
PROD_PRED_EXPL	SEQ_ITP	STATE	32	false	0	395	7390
PROD_PRED_EXPL	SEQ_ITP	DYNAMIC	32	-	-	336	12922
PROD_PRED_EXPL	SEQ_ITP	DYNAMIC	1	-	-	285	8404
PROD_PRED_EXPL	SEQ_ITP	ARG_WHOLE_PRED	1	false	0	247	13660
EXPL	SEQ_ITP	NO_OP	-	-	-	239	8575
PROD_PRED_EXPL	SEQ_ITP	ARG_T	32	false	0	239	11805
PROD_PRED_EXPL	CEX_BASED	NO_OP	-	-	-	220	11463
PROD_PRED_EXPL	SEQ_ITP	PATH_T	32	false	0	216	5840
PROD_PRED_EXPL	SEQ_ITP	ARG_WHOLE_PRED	32	false	0	195	5693
PROD_PRED_EXPL	SEQ_ITP	ARG_T	1	false	0	175	5523
PROD_PRED_EXPL	SEQ_ITP	PATH_T	1	false	0	162	6798

Table 4.8: All the measurements ran

Table 4.8 summarizes the results of every evaluated configuration strating from the best to the worst. While the predicate abstraction algorithm produced the best results, there is only a slight difference to the best product abstraction based strategies, which were the path- and ARG based algorithms without using $\top$ values, with limit 1.

In general configurations with lower limits had better performance and although with information sharing and the changing of the coverage checking order the performance did become a little better, the changes were insignificant.

Also, an interesting pattern can be found in the heatmaps of the measurements. Configurations with the low limit have generally better results in the program category plc, and slightly worse resluts in eca. At the same time, the opposite is true for configurations with the high limit: they have better performance in category eca and worse in plc.

The most important thing to note is that although the state-based strategy had an average performance, it did manage to verify a good number of model in category eca, which the predicate abstraction could not. Since explicit-value analysis had the best performance in that category too, these evaluations show that the combination of the two different domains was successful, we managed to incorporate the benefits ot both strategies into one.

Chapter 5

Conclusion

In our work, we presented seven 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 if the number of different values for a variable exceed a given limit, track a predicate containing this variable instead. 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 also created two additional strategies that do not track explicit values and predicates simultaneously, but rather switch between the two domains if the number of different values of a single variable, or of the whole ARG exceed a given limit.

We implemented our new strategies in the open source Theta verification framework. We also implemented two additional, already existing algorithms to be able to compare our newly introduced strategies with related work.

We proposed research questions and evaluated them by running 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 strategies using $\top$ values are less efficient, but the other algorithmsall outperform pure explicit-value analysis, and the path- and ARG-based ones with the best configurations can measure up to 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.

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.

It would also be important to analyze our results further and find an answer to why some configurations perform better in certain program categories. A through analysis could even reveal new patterns or characteristics that could lead to similar findings concerning software verification.

Bibliography

[1] Viktória Dorina Bajkai. Combining abstract domains for software model checking. Bachelor's thesis, Budapest University of Technology and Economics, 2018.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] Ákos Hajdu. *Effective Domain-Specific Formal Verification Techniques*. PhD thesis, Budapest University of Technology and Economics, 2020.

[24] Ákos Hajdu and Zoltán Micskei. Efficient strategies for CEGAR-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020. DOI: `10.1007/s10817-019-09535-x`.

[25] 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.

[26] 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`.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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