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

Improving Formal Verification with Portfolio-based Runtime Techniques

Scientific Students' Association Report

Author:

Zsófia Ádám

Advisor:

Zoltán Micskei, PhD

2021

Contents

Kivonat

A formális verifikációs módszerek lényege a matematikailag precíz reprezentációk és algoritmusok használata programok és modellek tulajdonságainak ellenőrzésére. Térnyerésük, különösen a biztonságkritikus rendszerekben, folyamatosan erősödik. Ugyanakkor a formális módszerek magas számításigénye miatt sok, különböző területre szabott algoritmus létezik.

Az esetek többségében megfelelő algoritmus és konfiguráció választása egy adott problémára komoly szakmai tudást igényel (például annak kiválasztása, hogy milyen absztrakciós módszert állítsunk be). Még egy szakértőnek is gyakran több konfiguráció kipróbálására van szüksége ahhoz, hogy találjon egy, az adott problémán jól teljesítő beállítást. Ugyanakkor a verifikációra rendelkezésre álló idő és erőforrások az esetek többségében korlátozottak.

Munkám célja olyan technikák ajánlása, amelyek a rendelkezésre álló idő hatékonyabb kihasználását segítik. Felteszem, hogy adott egy konfigurálható, szekvenciálisan futtatható eszköz, egy bemeneti probléma és egy időkorlát. Olyan módszereket javaslok, amelyek dinamikusan választanak és változtatnak az éppen futtatott konfiguráción ezzel egy összetett portfóliót képezve. A módszer újdonságértéke, hogy nem csak a bemenetből kinyerhető, de futás közben gyűjtött információkat is használ, hogy beavatkozzon az algoritmusba vagy éppen új konfigurációra váltson.

A javasolt technikákat megvalósítottam a Theta keretrendszerben, C programok verifikálására fókuszálva. Az eszközt kiegészítettem egy, az algoritmus megakadását felismerni képes futásidejű funkcióval, illetve egy ehhez jól illeszkedő portfólióval, amely változatos konfigurációk közül választ, ezzel is növelve a siker esélyét.

A kiértékeléshez az *International Competition on Software Verification (SV-COMP)* verifikációs problémáinak egy részhalmaza került ellenőrzésre. Az SV-COMP egy széleskörűen elismert, de-facto standard C nyelvű programgyűjtemény, kifejezetten szoftver ellenőrző eszközök számára. A javasolt javításokat néhány, általában jól teljesítő konfiguráció mérési eredményeivel vetettem össze, melyeket egyesével, illetve naiv szekvenciális portfólióban is futtattam. A mérési eredmények megmutatták, hogy az eszköz a hozzáadott technikákkal több feladatot képes megoldani, ráadásul lényegesen kevesebb idő alatt, mint a többi esetben.

Összefoglalva, munkám során olyan megközelítéseket dolgoztam ki, amelyek a hatékony verifikációt segítik a konfiguráció és algoritmus szakértői tudást igénylő kiválasztásának automatizálásával. Eredményeim bármely verifikációs keretrendszer fejlesztőjének segítséget nyújthatnak. A kiértékeléshez egy konkrét eszközön el is végeztem a módszerek megvalósítását, megmutatva, hogy milyen módon lehetséges ezek segítségével a teljesítmény javítása.

Abstract

Formal verification is an approach of using mathematically precise representations and algorithms to check properties of a given program or model. Formal verification is gaining increasing importance, especially in safety-critical domains. However, formal methods are computationally complex, which has resulted in various efficient algorithms tailored for different application domains.

In most cases, choosing the right algorithm and configuration for a given problem requires expert knowledge (e.g. which abstraction method to use during the verification). Even an expert might need to execute several configurations before finding one that performs well on the given verification task. But time and resources are limited in most cases.

My goal in this work is to propose techniques that help utilize the available time more efficiently. Assuming a configurable verifier tool, an input task, a time constraint and sequential execution, the proposed methods select and dynamically change verification configurations forming a complex portfolio. The novelty of the method is that it does not only rely on information from the input task, but also tracks runtime progress information from the verifier and intervenes in the current execution by switching configurations.

To show how these techniques can be tailored to a tool in practice, I realized them in the tool Theta, focusing on C program verification. Improvements include adding a runtime enhancement to the abstraction-refinement loop of the algorithm, which is capable of detecting when the algorithm is stuck. A portfolio offering a diverse set of configurations with algorithm selection is also added, such that it complements the runtime approach to increase the chance of success.

For evaluation, a subset of the benchmarking tasks of the *International Competition on Software Verification* (SV-COMP) is used. SV-COMP is widely regarded as a de-facto standard benchmark set of C program verification tasks. The recommended improvements were compared to several generally well-performing configurations, executed one at a time and in a naive sequential portfolio as well. The tool with the improvements proposed in this report proved capable of solving more tasks significantly faster than those in the baseline configurations.

To summarize, I designed approaches for efficient verification by automating some of the configuration and algorithm selection tasks requiring expert knowledge. The approaches are general and applicable in any verification framework. To evaluate them, I realized these techniques in a specific verifier framework, showing how it can improve the tool's performance.

Chapter 1

Introduction

Formal verification approaches the task of verifying software code and models using mathematically precise representations and algorithms. One of the most widely used formal verification techniques is *model checking* [34][4], which traverses through every possible execution with every possible input to verify if a given property holds. These techniques are becoming more and more important as the complexity of software increases in safety-critical domains, where there is a risk of not just financial loss but also injury of people in the case of an accident.

The *exhaustive nature* of model checking makes it able to prove not just the presence, but also the absence of *property violations* (i.e. errors of a certain type). However it also causes the disadvantage of being computationally expensive. Therefore several model checking algorithms have been proposed, such as symbolic methods [20], bounded model checking [15] or abstraction-based techniques [23][24]. Creating new or improving existing algorithms is still a widely researched and actively expanding topic.

Motivation There are many *model checking tools* under active development that are capable of verifying different *models or software code* written in different programming languages (e.g. tools participating in the annual *Model Checking Contest* [40] or the *International Competition on Software Verification* [9]). Most of these verifiers have several configuration options or even several algorithms implemented to be able to check a wide variety of input tasks [10]. For these tools it is crucial to be able to give their users at least some advice on what configuration to use, as some of their users might not have a deep understanding of the theoretical background of the algorithms and its connections to the application domain. Many of the tools already use some kind of *portfolio* (i.e. a strategically assembled set of configurations) or similar technique to tackle this challenge [42][27][46].

It is not uncommon that an algorithm is too slow, runs into a non-terminating execution or returns with an inconclusive result. The need for the already existing portfolios arises from the fact that the efficiency of the existing model checking algorithms depends greatly on the domain and different properties of the *given input.* Most of the already existing solutions stem from a single, algorithm-specific idea, growing further organically. But systematic planning is usually not a priority, which might result in suboptimal solutions.

Goal My goal in this work is to analyze and propose techniques that can help utilize the available time more efficiently with any model checker. For this I assume that there is a given time limit, the verification tool is configurable and the execution of these configurations has to be sequential. I will concentrate on proposing techniques that help in the creation of a complex and dynamic portfolio. To the best of our knowledge, currently there are no systematic collections or guidelines on portfolio techniques for model checkers, which I strive to change with this work.

Contributions In Chapter 3 I outline the general process of verification with a model checker. I go through this process step by step and extract information that can be used to create dynamic portfolios with *algorithm selection* [45][39]. I propose techniques using not just algorithm selection based on the input tasks [30], but also an approach of using *runtime information* about the running analysis to help or stop the current configuration and how to use these to design a *portfolio* of dynamically changing configurations.

In Chapter 4 I realize the general approaches of Chapter 3 in the tool Theta [49], a highly configurable model checking framework built around abstraction refinement-based analysis. Focusing on C program verification, I show how the general ideas can be applied in Theta. I assemble a dynamic portfolio, which contains my work 1) on *algorithm selection* to choose not just an initial strategy, but to make decisions about what configuration to use next throughout the analysis, 2) a *runtime algorithmic improvement* detecting when the explicit abstraction-refinement based analysis of Theta is stuck and guiding it out of that state or if that fails, stopping the configuration and choosing another one instead.

Evaluation To evaluate my work I ran benchmarks on a set of C programs from the *International Competition on Software Verification (SV-COMP)* [9], which is a standard set of C and Java benchmarks for software verifiers endorsed both in academy and in industry. The tool Theta has already taken part in last year's competition as the backend of the toolchain Gazer-Theta [1]. This time it is evaluated using its own C frontend and the dynamic portfolio, which is implemented directly as part of the tool rather than a separate script. The dynamic portfolio is compared to two generally well-performing configurations executed by themselves and to a simple sequential portfolio of three diverse configurations. The dynamic portfolio outperforms all of these in most cases both in average CPU time and number of tasks solved.

Conclusion To summarize, in this work I propose a method to systematically design and assemble dynamic portfolios for model checking tools to make verification more efficient. Based on this method I realized a dynamic portfolio in the model checking tool Theta, including algorithm selection and runtime improvements for abstraction-refinement based [23] configurations. I also evaluated this portfolio on C verification benchmarks to show how it can improve a tool's general performance. Any verification tool can benefit from these methods as they make the usage of the tool easier for a user without deep theoretical knowledge of formal verification.

Chapter 2

Background

In this chapter I would like to provide an introduction to *formal verification* and then focus on *software model checking.* Furthermore I define and describe a *formal representation* for programs (Section 2.2.2), *error properties* (Section 2.2.3) and an *algorithm* (Section 2.3) used during formal verification that will be necessary to understand the subsequent chapters. I also introduce how a tool's high configurability can affect its usability, with or without using *portfolios* and *algorithm selection* in Section 2.4.

2.1 Formal Verification

The swiftly expanding area of software and model analysis consists of broad varieties in static and dynamic techniques, which are capable of finding several types of issues. What makes *formal verification* techniques unique is their aim to not just find errors, but to provide mathematically precise proofs about the correctness of the input program or model.

Model checking is a formal verification method that proves the presence or the absence of errors in the input model through exhaustive traversal of every possible execution. For example, a model checker might accept an extended finite state machine as input, such as the simple example in Figure 2.1. The model checker tool will also need an error property to check, for example whether there are such input values that c is reachable *(reachability property)* or are there such input values that at some point the value of y becomes less than zero *(overflow property).*

The tool will then apply a formal verification algorithm. It might traverse the graph with a simple DFS [7], but that will not be enough to conclude anything as the possible values of the variables throughout all the possible steps taken in the model will also have to be taken into account. This introduces the problem of *state space explosion*, meaning that even if x and y are simple 32-bit integers, with each such integer we get a multiplier of 2^{32} on the number of possibilities.

Tackling state space explosion is an issue which has been extensively researched and there are a wide variety of techniques available, such as symbolic methods [20], bounded model checking [15] or abstraction [23][24].

After executing the algorithm, the tool outputs if the property holds or not and it might even give a counterexample or a proof to support the result.

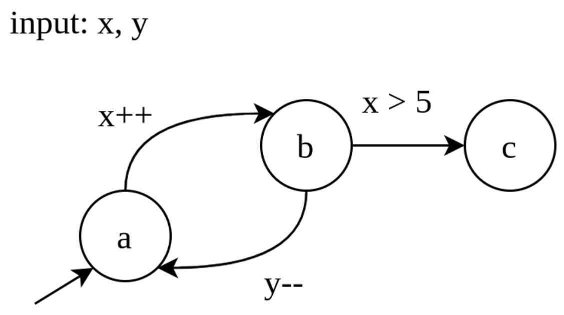


Figure 2.1: A simple extended state machine of three states and two input variables

Definition 1 (Verification Task). A verification task, i.e. an input task for a model checker consists of two parts:

- an *input model* (or program code) to be verified by the model checking algorithm,
- an *error property* to be checked by the model checking algorithm on the input model

The result given for the verification task can be *inconclusive* (with other words, *unknown*) or if the algorithm is successful then either *safe*, if the error property holds or *unsafe* if it does not hold.

If the result given by the algorithm is wrong due to some kind of error than the result can be called either *false positive* (if the result was *unsafe*, but it should have been *safe*) or *false negative* (if the result was *safe*, but it should have been *unsafe*). ▪

2.2 Model Checking Software Code

Software model checkers work in a similar way as the example above in Section 2.1, but instead of getting a model as input, they accept software code written in programming languages they support. As this work is focused mostly on the verification of C programs, a simple C program is given below, which will be our running example during this section as we introduce a formal representation, typical error properties and a verification algorithm used by software verification tools and used in this work as well.

2.2.1 A Simple Example

The code shown in **Listing 2.1** has a function declaration without definition (*__VERIFIER_nondet_int()*), which we assume can return any valid integer. This simulates some kind of user input or a message from another process and so on – the point is, that we only get the concrete value during execution. The function *reach_error()* also has a special meaning: reaching the call of this function is an error and it is unsafe, so it should never happen in a safe program. This could also be subsituted with an *assertion*, which would not have to be an annotation added for formal verification, but a statement that is often used by the programmers themselves.

Other than this, the program is simple and showing that it is safe is easy enough: we could only reach the body of the inner *if statement* if *x* would be greater than *0* and exactly *0* at the same time, which is impossible.

```c
extern int __VERIFIER_nondet_int(void);
void reach_error() {}

int main() {
    int x = __VERIFIER_nondet_int();
    if (0 < x && x < 5) {
        if (x == 0) {
            reach_error();
        }
    }
    return 0;
}
```

Listing 2.1: An example input of a verifier tool

2.2.2 Control Flow Automaton

Reasoning with mathematical precision requires a formal representation, even if the input originally was not in that format. In the latter case, the input has to be transformed to this representation and at the end the result has to be projected back to the original input (e.g. in which lines do the steps of the given counterexample happen). For the verification of C programs the *Control Flow Automaton (CFA)* [13] is a practical formal representation to use.

Definition 2 (Control Flow Automata). A CFA is a 4-tuple (L, E, l_0, l_q) where

- $L = l_0, l_1, l_2, ...$ is a set of locations, modeling the program counter,
- l_0 is the initial program location, the entry-point of the program,
- $E \subseteq L \times Ops \times L$ is a set of edges, which represent the executed operations between two locations. The above operations (Ops) can either be assignments (e.g. $x := y+2$) or assumptions (e.g. $[x = 0]$). ▪

In a CFA, such as the one in Figure 2.2, program executions correspond to paths in the CFA. It is important to note the difference between a location in the CFA and in a state of the program. The locations do not store information about the current values of the variables, but these are by all means part of the states of a program.

2.2.3 Reachability Problems

We also need to precisely define the errors we are looking for. Getting back to the example of verifying C programs, there are several possible properties, such as the reachability of an error location (as in the case of Listing 2.1), the ability to terminate or the possibility for variables to overflow. The format of these properties can be textual, but often they are formulated with some kind of temporal logic, such as linear temporal logic (LTL).[1]

In this work we will focus on reachability problems. Error states and error locations come quite naturally both in models and software, e.g. assertions, which check if a boolean property is true in a certain point of the execution or error handling functions, which are called if the system reaches a certain error state or a certain point in a program, which shouldn't be reached. A verification tool can prove if given function calls or failing assertions are possible in any execution of the program or not.

[1]Examples of typical properties in a textual and LTL format under the Properties section: https://sv-comp.sosy-lab.org/2022/rules.php

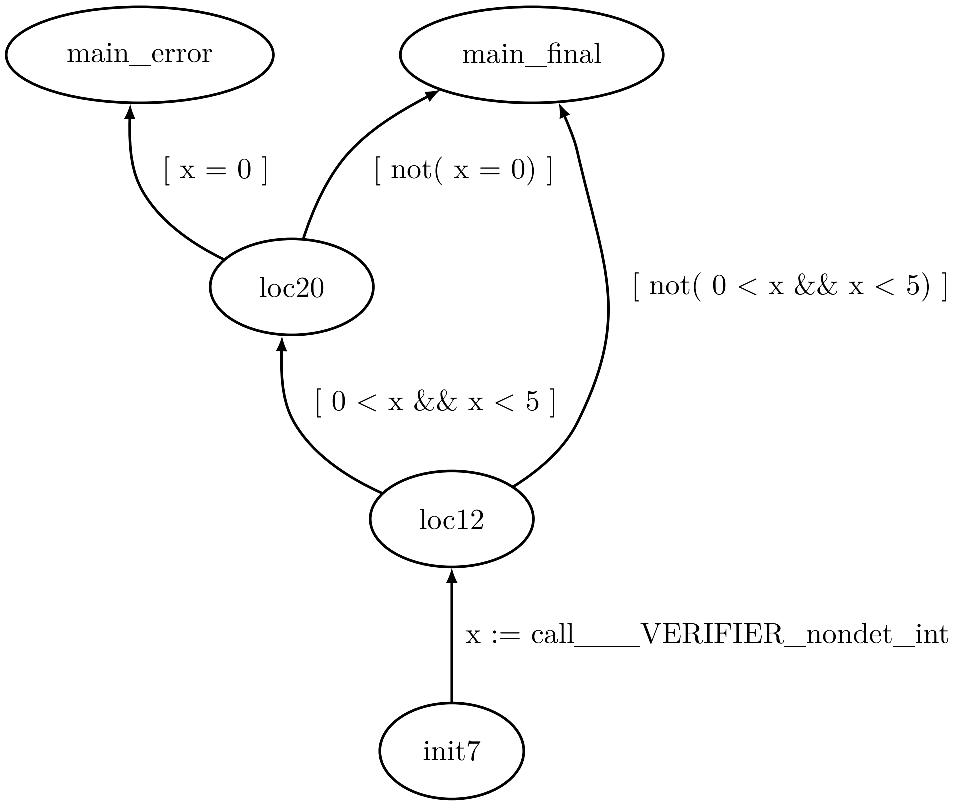


Figure 2.2: This *Control Flow Automaton* represents the C Program given in Listing 2.1. The initial location represents the starting line of main and on the first edge the return value of the first function call is assigned to *x*. The assumptions on the outgoing edges of *loc12* and *loc20* represent the branches of the if statements and in the end we can either reach a *final location* or an *error location*. This model is a simplified version of what the tool Theta uses for verification. The flow of control is the same, but the ones used in the tool have some more nodes and edges to handle important properties of the C language, such as value boundaries of integers.

2.3 CEGAR-based Model Checking Algorithms

Counterexample Guided Abstraction Refinement (CEGAR) [23] is a widely-known algorithm of using repeated *abstraction* and *refinement* steps of the state space for model checking. Using abstraction the model checker does not have to traverse all the possible execution paths for each input, rather many of these can be merged into a much smaller number of *abstract states.*

2.3.1 Abstraction

Abstraction is CEGAR's remedy against state space explosion. In this work we will use two possible ways of creating abstract domains over the variable values.

Explicit-value abstraction [11] utilizes the fact that there might be variables that are irrelevant to the safety of the model and the model checker can ignore these. It introduces a special *unknown* value that is assigned to untracked variables or tracked, but unassigned variables. This way it prevents the enumeration of all possible values.

Predicate abstraction [33] Proving something is often possible knowing only that a variable's value is smaller or greater than a constant and the concrete value of the variable itself is superfluous. Using this, Predicate abstraction stores predicates like $x < y$ or $x >= 5$ and tracks whether these hold or not.

The abstract state space created with either of the above domains is represented using an *abstract reachability graph* (ARG) [12].

Definition 3 (Abstract reachability graph). An abstract reachability graph is a tuple $ARG = (N, E, C)$ where

- $N \subseteq S_L$ is the set of *nodes*, each corresponding to an abstract state in some domain with locations D_L.
- $E \subseteq N \times Ops \times N$ is the set of directed *edges* between locations, labeled with operations. An edge $(l_1, s_1, op, l_2, s_2) \in E$ is present if (l_2, s_2) is a successor of (l_1, s_1) with op.
- $C \subseteq N \times N$ is the set of *covered-by edges.* A covered-by edge $(l_1, s_1, l_2, s_2) \in C$ is present if $(l_1, s_1) \sqsubseteq_L (l_2, s_2)$. ▪

In Figure 2.3 there are two ARGs based on the CFA of Figure 2.2 with different precisions: the one on the left tracks the predicate $x > 0$ and this predicate is already enough to eliminate the possibility of reaching the error location, thus it already proves the safety of the model. The one on the right tracks the predicate $x < 5$, which is not enough for a proof in itself and thus the error location is present in one of the abstract states.

An abstract state can represent many or even an infinite number of concrete states – but the more concrete states represented the more is lost in terms of precision. By definition abstraction means that we lose information and thus tracking values only of these abstract domains is an *over-approximation*, meaning that we can get *false positive* outcomes, as depicted in Figure 2.4.

Figure 2.4 shows how the *abstract counterexample* might hide an infeasible path in the input task. If we need to find a path from the light to the dark concrete states then from

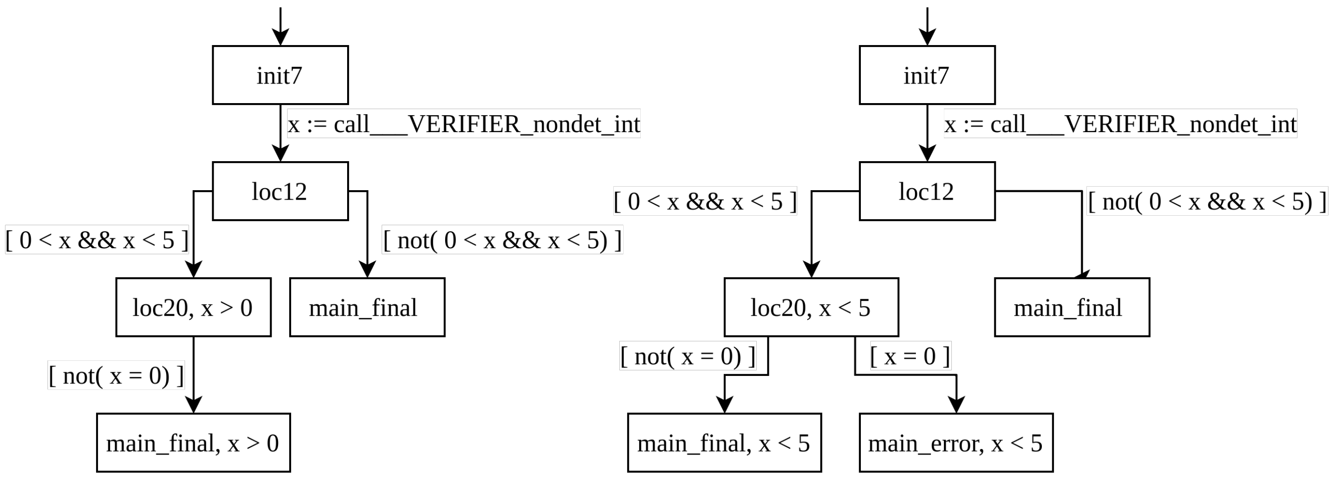


Figure 2.3: Two possible ARGs built from the CFA on Figure 2.2. The rectangles represent the abstract states. The left tracks the predicate $x > 0$, while the right ARG tracks the predicate $x < 5$.

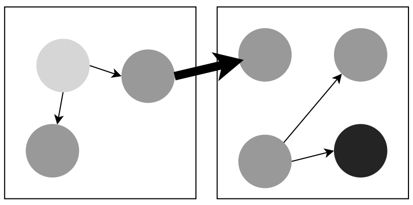


Figure 2.4: The circle nodes represent concrete states, whereas the rectangles are abstract states so, that they represent the concrete states that are in their area.

the abstract perspective of the rectangle it might seem possible through the thick arrow, even though in reality there is no path between the two. To handle these false positives, refinement is introduced.

2.3.2 Refinement

Abstraction will output an *(abstract) counterexample*, which might not be feasible in the concrete model. In the next step, the *refinement algorithm* checks the abstract path's feasibility. If it is infeasible, the precision of the abstraction is refined by inferring new variables or facts to be tracked, and the ARG is pruned to exclude the spurious counterexample.

Definition 4 (Counterexample). An abstract counterexample is a path in the ARG from the initial state to a state containing the error location. $\sigma = ((l_1, s_1), op_1, \ldots, op_{n-1}, (l_n, s_n)) :=$ path to unsafe node (with l_E) from ARG

Checking the feasibility and refining the precision are both carried out with the help of an SMT solver.

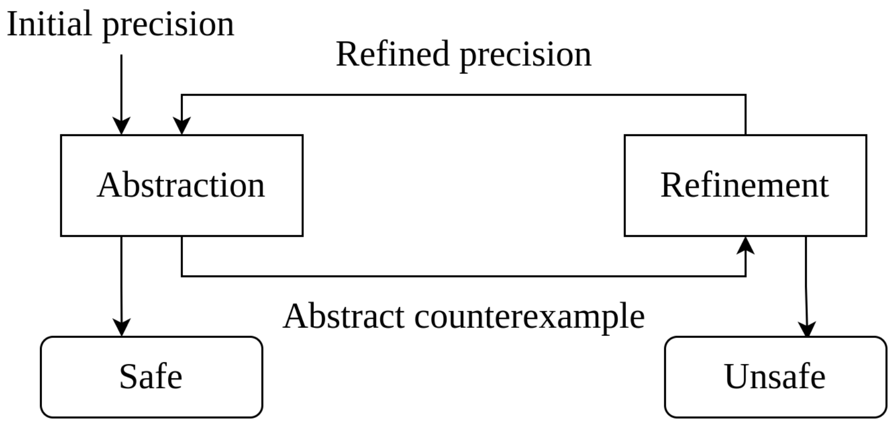


Figure 2.5: The CEGAR loop: when a possibly false abstract counterexample is found,the refinement algorithm checks, if it hides a feasible counterexample or not. If it does, the program is unsafe and the verification terminates. On the other hand, if we cannot find any abstract counterexamples anymore then the program must be safe.

Propositional logic In propositional logic [18], a *formula* is composed of Boolean variables and connectives (such as $\neg, \vee, \wedge$). An *interpretation* I assigns each variable a truth value (true or false). Given a formula φ and an interpretation I we say that $I \models \varphi$ (I "models" φ) if φ evaluates to true under I. A formula φ is *satisfiable* if an interpretation I exists with $I \models \varphi$.

First-order logic First-order logic (FOL) [18] generalizes and extends propositional logic with predicates, functions, and quantifiers. A formula φ is satisfiable if an interpretation I exists with $I \models \varphi$ [18].

Definition 5 (Satisfiability modulo theories). The *satisfiability modulo theories* (SMT) problem [7, 17] is to decide if a formula φ is satisfiable in a theory $T(\Sigma_T, A_T)$. ▪

Church [22] and Turing [51] proved that satisfiability is undecidable for FOL in the general case. However, in practical applications, the problem is often decidable because there are different background theories, which give particular meaning to predicates and functions, and restrict the signature and the usage of quantifiers. SMT solvers are created using many possible techniques and optimizations to solve these problems.

2.3.3 The CEGAR Loop

CEGAR algorithms utilize abstraction and refinement in a so-called CEGAR loop (Figure 2.5). The loop alternates between the abstraction and refinement algorithms, constantly pruning and rebuilding the ARG while refining the precision more and more until either refinement finds a feasible counterexample or abstraction builds an ARG, where the error location is unreachable.

2.4 Verifying Software in Practice

2.4.1 Utilizing configurability with Portfolios and Algorithm Selection

Verifying complex software with a software verification tool often requires special knowledge about the techniques applied by the tool. The effectiveness of a given verification algorithm varies greatly depending on a broad variety of the input program's properties (e.g. presence of loops, cyclomatic complexity, number of variables), which are not necessarily known, if not explicitly analyzed. But even if they are, the user also has to know, how to configure the tool based on this information.

But configurational possibilities vary greatly from tool to tool, although there are some general options that are often available, such as *timeouts*, boundaries of usable hardware resources e.g. *memory consumption, number of CPU cores.* A typical example of a not so general option ensuing from the technique used is the choice between abstract domains, such as the *explicit* and *predicate domains* introduced in 2.3.1.

A *portfolio* is simply a given set of model checking algorithms or configurations, preferably with a way to execute these either sequentially or in parallel. A diverse, but static *portfolio* of a few configurations can greatly increase the general usability of the tool [1].

On the other hand, it is also possible to automate choosing a preferable algorithm or configuration based on the input model or program – this method is called *algorithm selection* [45][39]. Each of these methods help in utilizing the tool's features better, preferably in an automated way.

2.4.2 Configuring CEGAR

The open source verification tool Theta [49] is a configurable model checking framework built around CEGAR [23]. It is capable of checking several formal representations and has a growing number of configurational possibilities. It is capable of transforming C programs to a format called *eXtended Control Flow Automaton (XCFA)* and to *CFA* as well, thus it is capable of C software verification.

In the next sections a summary of several configuration options of Theta are given, which will play an important part in this work.[2]

Arithmetics By default Theta uses integer arithmetics to handle the variables in the input program, i.e. using mathematical (unbounded) integers. But if the input task contains floating point values or bitwise operators then it is necessary to change this to a bit-precise arithmetic, which can handle these as well, although it is much less efficient and cannot apply refinement techniques using interpolation.

Abstract Domains An important configuration option, abstract domains, were already briefly introduced in Section 2.3.1. In practice, Theta has eight domain options currently (but only four can be used on CFAs), although they are all based on the two domains introduced earlier.

Three of these are predicate abstractions and they mostly differ in the format of predicates they use. One uses Cartesian predicate abstraction and conjunctions of logical

[2]Documentation of these options: `https://github.com/ftsrg/theta/blob/master/doc/CEGAR-algorithms.md`

predicates. The other two uses Boolean predicate abstraction. The remaining one is an implementation of the explicit abstraction introduced earlier.

Refinement Possibilities In theory, a refinement strategy can be as simple as adding a new variable to the precision in each iteration, but usually refinement algorithms aim to help the convergence of the analysis to a result be as fast as possible.

In practice most refinement strategies use SMT solvers, mostly with *interpolants* [43] (e.g. the refinement options *BW_BIN_ITP, SEQ_ITP*) to try and find variables or predicates that are worth to add to the precision based on the given infeasible counterexample.

Initial Precision Generally a CEGAR analysis can start with an empty precision, extending the precision iteratively later. But in Theta it is also possible to traverse through the input formalism and collect all the variables or predicates that it contains, adding them to the initial precision. These configuration options are called *ALLVARS* and *ALLASSUMES*.

2.4.3 Executing Theta on the Simple Example

A trivial example to why utilizing the configuration options of a tool is important can be given just by executing the tool Theta on the running example C program of Section 2.2.1. It is easy to see, that the program is safe. But if we choose a configuration using explicit domain, we can easily run into a never-ending execution of the tool.

In the first line of *main* the variable x is marked as unknown and in the case of an unknown value both conditions in the if statements are satisfiable and so we remain in a continuous state of not enough information to prove anything. This problem can be solved by changing the configuration in the following ways:

- we can just change the domain to another one, e.g. to *Cartesian predicate abstraction*,
- or we can use a special algorithmic improvement of Theta, the configurability of the explicit domain and use this option to let Theta enumerate a limited number of values for x

In these latter cases Theta solves our problem in seconds without much need for hardware resources (see **Table 2.1**). Of course this example is by no means a precise benchmark, but a simply explainable example of why the diverse configurations of Theta are important to use.

Time to solve	Domain	Max. num. of enumerated states
-	explicit	1
around 1s	cartesian predicate	-
around 1s	explicit	10

Table 2.1: Theta is incapable of solving the example in the explicit domain, without enumeration. Changing any of these options solves the issue.

Motivation After introducing even just a few of the existing formal methods, it is clear that there is no absolute winner when talking about formal verification techniques. Due to this, many existing tools implement diverse configuration options and algorithms and many of them also use some kind of individual technique to utilize these (e.g. a portfolio). In this work, I would like to propose possible techniques for this on a more general level, so to give a guide on utilizing a tools' already existing components with the help of complex and dynamic portfolios.

Chapter 3

Improving the Efficiency of Verification Tools Using Dynamic Portfolios

This chapter proposes a systematic design technique of *algorithm selection* and *portfolio assembly* for formal verification tools. This technique is described in a general, tool-independent manner, thus it can be applied to any model checker.

After the assumptions and goal of this work is formulated (Section 3.1), a *high-level verification process* is introduced, i.e. what general steps are executed when a model checking tool is used (Section 3.2). Then I describe a *simple portfolio pattern* (Section 3.3). Next I systematically collect the possible sources of information throughout this process to be used for algorithm selection (Section 3.4) and lastly I propose a technique for the assembly of *dynamic portfolios* (Section 3.4.4).

3.1 Assumptions and Goal of the Work

There is no single best algorithm or configuration in formal verification [9] . This implies that verifiers will either have several algorithms or at least several configurations or configurable properties at hand [35]. The next step of progress for a tool is *the automation of algorithm selection* or at least the automatic usage of multiple configurations, i.e. a *portfolio.*

Assumptions In this work we consider the following constraints of the tools:

- the tool has configurable properties *(e.g. is capable of several algorithms or variations of an algorithm)*
- the tool can only be executed sequentially *(no concurrent executions)*
- a (CPU) *time limit* is given
- we try to optimize primarily to succeed in solving as many tasks as possible *(each task within the given time constraint)* and secondarily to do this in the shortest possible cputime

These constraints can be easily modified to meet other requirements or the lack of some (e.g. it might be realistic to consider a given number of parallel executions and that we are optimizing for walltime and not CPU time – in that case a sequential portfolio should be made into a paralell one), but these constraints offer a convenient place to start.

Based on related work on this topic the typical solution for portfolios starts out from a single idea based on the tool itself and then they evolve further organically, which might not result in the most efficient solution.

Goal The goal of this work is to give a general process of dynamic portfolio design, which enables the systematic planning of portfolios so that all the available features of the given tool are utilized in an efficient manner.

Research Method First I started by reviewing the related work on algorithm selection and portfolio topics. Next I analyzed the state of the art tools and their portfolio solutions. Moreover the main source of this chapter is the *general experience* collected while working with the *frontend* and *portfolio* features of the tool *Theta* [49], both directly in this work and in the work published on SV-COMP 2021 [1]. The tool specific parts of this work are added in Chapter 4. To synthesise the knowledge gained, I systematically collected the choices of static and dynamic portfolio design.

3.2 The Process of Verification

The execution of a formal verifier typically consists of the steps depicted in Figure 3.1.

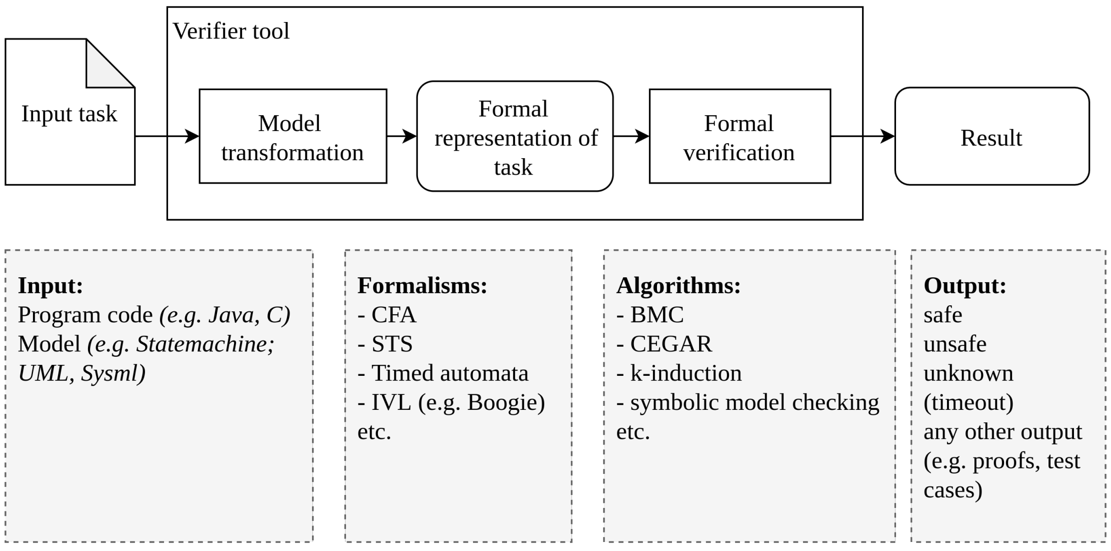


Figure 3.1: The typical process of formal software verification, including a frontend (model transformation) and a backend (formal verification).

The Input Task and the Frontend The process starts with some kind of input task – such as a *model or program code*. This is typically a product of an engineer's hard work and is fairly high-level or at least not formal. Thus in most cases it has to be transformed into a mathematically precise representation, which is done in the model transformation step. A *typical formal representation* for software code, the *Control Flow Automaton (CFA)*, was introduced in Section 2.2.2, but there are many other possibilities, for example *intermediate languages* [6, 32] or other kinds of automata [21].

The Backend and the Result The transformation to the formal representation concludes the *frontend* and the tool can now execute the verification itself in the *backend*. The number of the possible algorithms and their variations are ever growing. Well-known examples include *CEGAR* [23] (introduced in Section 2.3), *bounded model checking (BMC)* [16] or *symbolic model checking* [19]. In practice these algorithms typically have at least a few configurable steps, such as the abstract domains of CEGAR and a verification tool might be capable of executing not just one, but many of these algorithms.

If the analysis is successful then we get a *safe* or an *unsafe* result from the verifier. But model checking algorithms are computationally complex and a non-terminating execution is always possible, so generally a *timeout* is set to stop the tool after a while. If this or any other error occurs then the result is *unknown* or with other words, *inconclusive*. But verifiers might also be capable of giving other forms of output as well, such as a *proof of safety*, typically in the form of an automaton with invariants or a *counterexample* showing that the input is unsafe. This might be human-readable, executable or might be created to be validated by other tools[1] [8].

In the next sections we approach this process with the intention of injecting further steps, which make the tool perform better using the tool's already existing features.

3.3 Designing Sequential Portfolios

The simplest way of using multiple configurations is a *sequential portfolio* [1]. A few chosen configurations are placed one after the other so that they fit into the given time window. They are executed one after the other until one is successful or until we run out of time or configurations. Although this sounds simple, there are already some strategic decisions to be made:

- When should the next configuration of the queue start?
- Should there be an inner timeout for each configuration or should we just wait for the configuration's success or failure and potentially let it consume the whole time window?
- What happens, if a configuration fails earlier, how will the inner timeouts change then?

In Figure 3.2 the possibilities of a sequential portfolio are decomposed. The two main points to plan out are the following:

Configurations A small set of diverse *configurations* and the execution *order* of these has to be chosen.

Time limits Although a global time limit is already given, *initial local time limits* can be set inside the portfolio. It is also important to decide on what happens, if a configuration fails before it reaches its time limit.

[1]For example the standard format of violation and correctness witnesses used by the validators on SV-COMP: `https://github.com/sosy-lab/sv-witnesses`

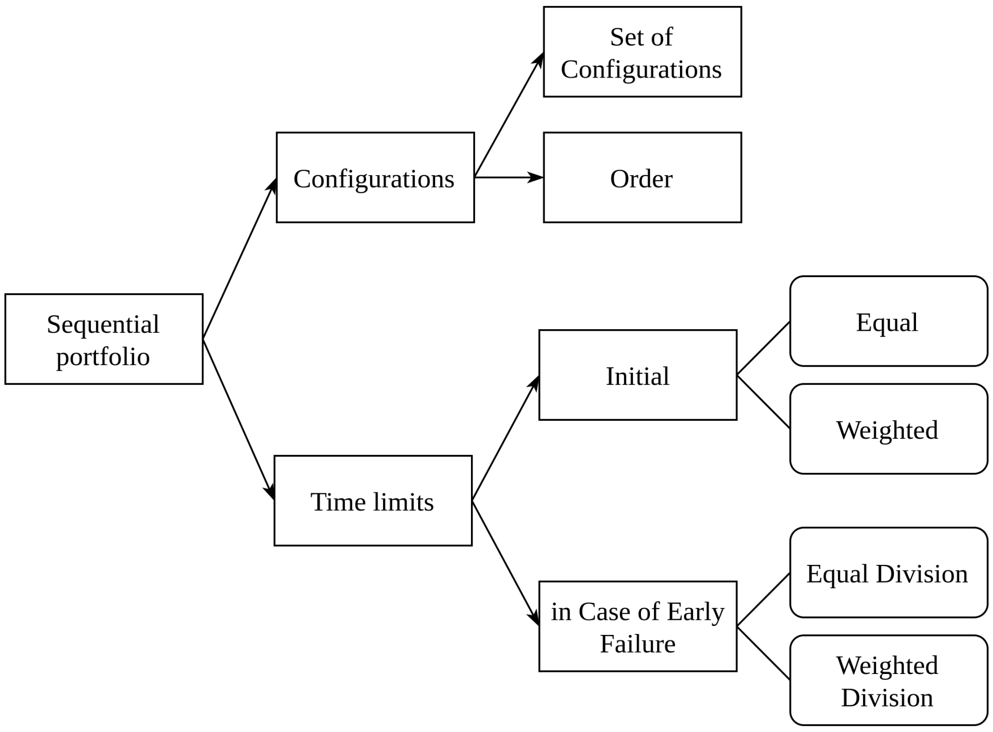


Figure 3.2: Possibilities in sequential portfolio design. Rectangles depict the topics of design choices, while the rounded rectangles are the possible solutions.

3.3.1 Good Practices for Sequential Portfolio Design

Advice on Configurations A sequential portfolio requires generally well-performing operations and even the number of those should be limited. Special configurations tailored to corner cases should not be added – too many small steps can easily slice the available time up and it will make the result ineffective.

Advice on Local Time Limits Although local time limits have to be thought through on a case-by-case basis, but a few useful points can be given generally:

- Verification tools are prone to timeouts and these timeouts can happen due to a bad configuration – if this is a possibility then local timeouts are crucial.
- The length of the local timeouts might be equal, but it isn't necessarily the best solution. For example a configuration, which is mostly fast when successful, but prone to timeouts, when unsuccessful is probably not worth to let run too long. In that case *weighted timeouts* have to be added, giving only a smaller portion of the time to the configuration mentioned above.
- In case of an early failure of a configuration, it might be worth to recalculate the timeouts based on the remaining time, so the other configurations do not gain unwanted advantage (see Section 3.3.2).

3.3.2 Examples for Sequential Portfolios

Let's say that we chose and ordered three different *configurations* to be put into a sequence. We also chose the *local time limits* to be equal, as there are no outstandingly slow or fast configurations; we simply chose these to cover as many diverse tasks as possible.

There is only the *case of early failure* left undecided based on Figure 3.2. What should we do if the first Configuration fails early? In Figure 3.3 we see some possibilities for timing: in P1 and P2 either *Configuration 2* or *3* gets advantage and that is probably unwanted, as we said, that the configurations in this case are about equally efficient. In P3, the remaining time is halved, so both configurations get an equal amount of extra time. This solution already points into a more dynamic direction (as it has to divide the remaining time during analysis).

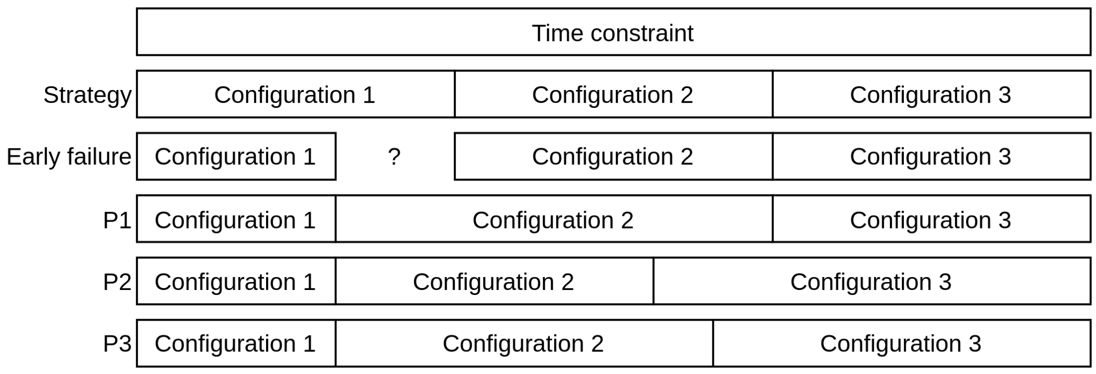


Figure 3.3: We have three configurations with equal local time limits. If *Configuration 1* fails early, how should the remaining time be utilized? The last three rows represent possible solutions.

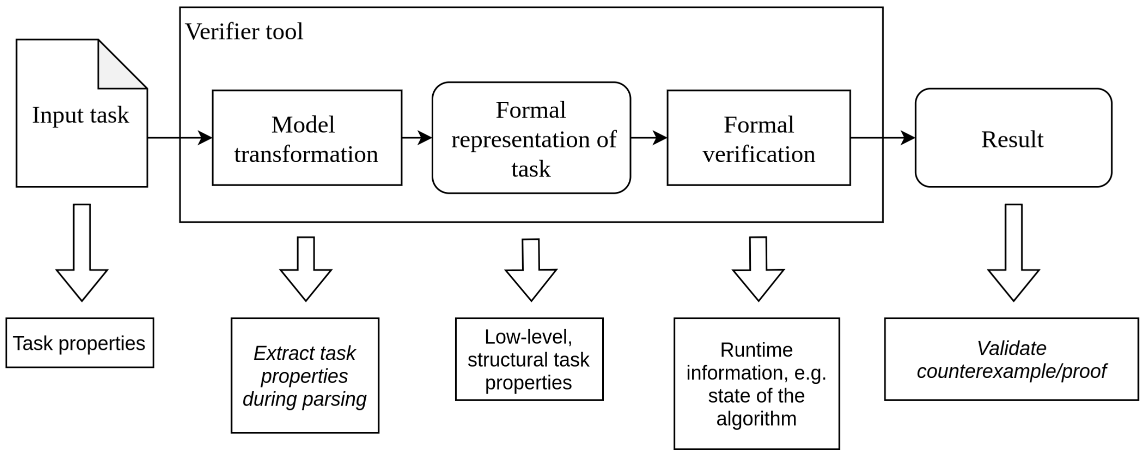
The above points will also be used in later chapters: different local timeouts are added to the complex portfolio in Chapter 4, whereas the baseline sequential portfolio of the evaluation uses the above mentioned recalculation of remaining time in Chapter 5.

3.4 Designing Dynamic Portfolios

The portfolio described above will work without any further additions and it is a good example of the usage of different time limits and configurations. However there is much more information that can be used in a portfolio, i.e. not just earlier experiences to create static configuration queues, but also information extracted before and during verification to be used in dynamic steps.

If we go through the steps in Figure 3.4, we can find information to use in every step.

- The formal representation and the original format of the task can both be parsed for different semantic and structural information.
- During execution different aspects and artifacts might be monitored that can give some kind of idea about the state of the analysis.
- In an ideal situation *wrong results* are nonexistent, but it isn't always the case – thus it might also be worth to check the *output results* and *validate* it if that is possible.



Verifier tool
Input task
Model transformation
Formal representation of task
Formal verification
Result
Task properties
Extract task properties during parsing
Low-level, structural task properties
Runtime information, e.g. state of the algorithm
Validate counterexample/proof

Figure 3.4: Throughout the analysis there are various forms of information that might be used in a portfolio

3.4.1 Dynamic Possibilities in a Portfolio

With the addition of the extracted information (Figure 3.4) a whole new set of possibilities appears in terms of portfolio features, as shown in Figure 3.5.

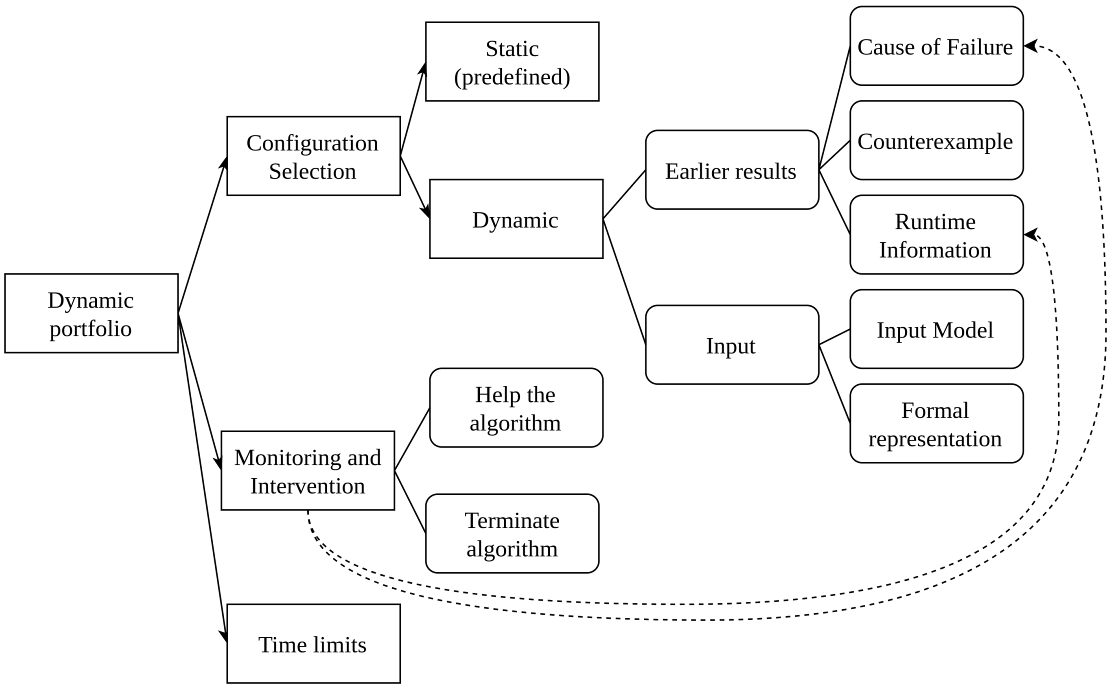


Figure 3.5: Possibilities of a dynamic portfolio. Rectangles depict the topics of design choices, while the rounded rectangles are the possible solutions. Time limits and Static configuration selection were already detailed on Figure 3.2.

Up until now the next configuration was *statically* selected from a *predefined queue*, but now it can be dynamically decided during the analysis based on information that was not available before execution. We will call this *dynamic configuration selection*. These dynamic decisions can be made based on the input task – the well-known name of this technique is algorithm selection [45]. This branch of Figure 3.5 will be detailed in Section 3.4.2.

On the other branch we find the terms *monitoring and intervention* – this refers to algorithm monitoring techniques that might intervene in the running configuration and which can also supply additional runtime information on a failed configuration to be used to decide on what configuration to execute next based on *earlier results.* These branches are described in more detail in Section 3.4.3.

Tailoring these techniques to a tool and putting them together into a *complex portfolio* might significantly improve its performance. Whereas the sequential portfolio has already built on the developer's theoretical knowledge and experience, here it will be utilized even more exhaustingly. This is presented on an example in Section 3.4.4.

Another Solution: Machine Learning instead of Manual Assembly Although in this work the main focus was manual assembly of portfolio features based on engineering knowledge, machine learning techniques can help or substitute several steps of this process. For example, searching for properties to base algorithm selection on can be done with *feature selection* techniques [50][25]). The step of manual assembly can also be realized with *machine learning techniques* instead. Usually for this some kind of *feature vectors* are created and the machine learning problem and algorithm is then formulated based on those, usually as a *supervised learning problem.* A common solution is to use the *support vector machine (SVM)* algorithm [30][50].

3.4.2 Configuration Selection based on the Input

Algorithm selection is a complex task and to add algorithm selection as a feature, connecting knowledge about the possible configuration's strengths and weaknesses with the *properties or property combinations* of the input tasks is necessary. The first step to realize that is finding the characteristic properties, like whether the input program contains loops or not.

The possible solutions are always heuristics in the sense that all possible tasks and property combinations cannot be known, so to some extent the portfolio will always be tailored for types of tasks that were available when creating it.

What we propose is to always carefully consider every piece of information that might be worth extracting from the input program or model. There is already plenty of work available on the possible properties. For example, empirical properties on variable types [29][3], control flow and loops [30] of C programs – this grouping of properties can be generally helpful in many situations, as it highlights such general constructs that can become problematic obstacles for any verification tool. There is also available work on identifying exactly what structural program properties might be important when choosing C verification tools with machine learning techniques [50].

Examining the Formal Representation The *formal representation* might also be worth to check for *properties.* Although at first glance it might seem like that analyzing two representations of the same model or program is superfluous, it is not. Using the formal representation it might be easier to deduce connections between certain configurations and the properties, as the model checking algorithms are executed directly on this representation. In either case, the *model transformation* step offers the perfect opportunity to extract these properties, as it involves parsing the input task and building the formal representation.

For example, when verifying software, the CFA is on a much lower level and its structure can be easily examined (e.g. *path lengths* or *cyclomatic complexity*). On the other hand, a program typically has semantic meaning added, such as the usage of different *variable types* or the usage of either a *for* or a *while loop*.

```
int i;
while (i < 10) {
        i++;
        ...
}
```

```
for (int i = 0;
i < 10; i++) {
        ...
}
```

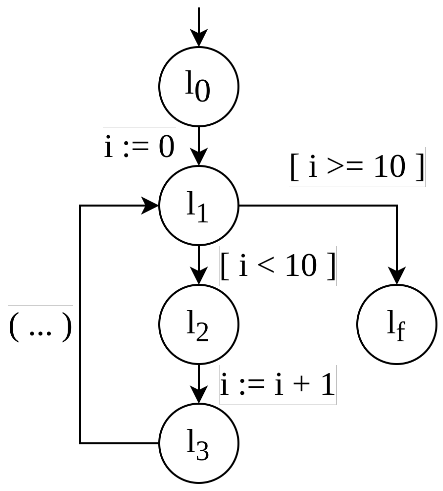


Figure 3.6: The same CFA loop can easily be a valid model of a for or a while loop as well

In Figure 3.6 there is an example of this semantic information: it is much more likely in the while loop that *i* is not a simple counter, but a more general flag that might be changed in any ways in the body of the loop thus creating a loop with an uncertain number of iterations instead of the fixed number of 10 iterations. Of course, to make sure about this, further analysis on *i* should be carried out, but differentiating between *for* and *while loops* can already be used as a simple heuristic in itself.

3.4.3 Monitoring and Intervention

In *bounded model checking* the analysis has to be given a *maximal bound* [16]. When the length of the paths observed by the algorithm reaches this bound, the analysis stops with an inconclusive result. Although this is normally seen as a necessary part of the basic BMC algorithm, it is also an example of a dynamic decision based on runtime information about the state of the algorithm.

If we are to design methods similar to the example above, we might want to consider the following:

- what is being built and stored during the algorithm – such as the *paths* explored in BMC or an *Abstract Syntax Tree or Graph* or *(abstract) counterexamples* or any other collection or function connected to the state space or the input,
- what *steps* does the algorithm consist of; if there are iterations, what steps do those have and
- are there any *particular states* of the algorithm that should be evaded (e.g. a state the analysis gets stuck in) or is particularly desirable (besides the trivial case of being successful).

Based on this information it might be possible and desirable to intervene in certain cases during execution. In the case of *BMC* this intervention meant *stopping* the analysis, but if there is a possibility to somehow help the algorithm, e.g. to help it out from an undesirable state, so it does not get stuck or to help it converge faster.

If the algorithm cannot be helped and it is stopped, one might also want to decide, what algorithm or configuration should be executed next – this can be decided based on, among other things, the circumstances of the intervention, e.g. if it was stopped due to a state space explosion or anything similar then are there any configurations that might prevent this from happening?

When implementing such ideas, one has to monitor the execution – this most likely means injecting checks inbetween certain steps in the algorithm and it might also create the need to store additional data, e.g. from earlier iterations. It is important to take into account the added overhead in time and in storage as well.

3.4.4 Assembling a Dynamic Portfolio – A Detailed Example

Algorithm selection chooses an algorithm directly before verification [39]. A portfolio, on the other hand, does not need to be more than a group of configurations that might be worth to use. If these techniques are mixed and decision points are added throughout the whole portfolio, the result is a dynamic portfolio able to include and exclude configurations during verification.

The earlier introduced techniques and properties fit into such a dynamic portfolio well. This is also showed in Figure 3.7. The upper diagram is a simple sequential portfolio similar to the example in Section 3.3 – consisting of three configurations, one after the other. However the one below employs dynamic techniques such as 1) choosing between *C3* and *C4* based on the input task, 2) monitoring the output of *C1* and choosing between *C2* or the *C3/C4* branch, 3) the special failure can be the result of a runtime monitoring method stopping the algorithm.

The advantages and possible reasoning behind this portfolio are the following. There is a maximum of 2 configurations executed sequentially, giving a chance to larger tasks that require more time.

- If the *special failure* usually happens quickly, *C2* might receive all the remaining time to itself – this is advantageous if it is a usually slowly converging configuration that can eliminate the *special failure* (an example of this will be given in Section 4.4.1);
- or, if the special failure happens after a long time, *C2* might be a usually fast "second chance", hopefully solving those tasks in a short time.

The *other failure* branch covers all the other possible non-successful outputs: local timeout, an unknown technical issue or any other possible failures. In these cases, *C2* might not be the best option (as *C2* was specialized for eliminating the *special failure*, but might not be a really good performing configuration in general), so *C3* and *C4* remain. *C3* might perform well only when a few general task properties match (or *C4* might not support some other properties), so here algorithm selection is utilized to decide on what to use next.

There is the obvious disadvantage of not trying *all* configurations from *C1* to *C4* and as almost all decisions are based on heuristics, there can always be corner cases that might

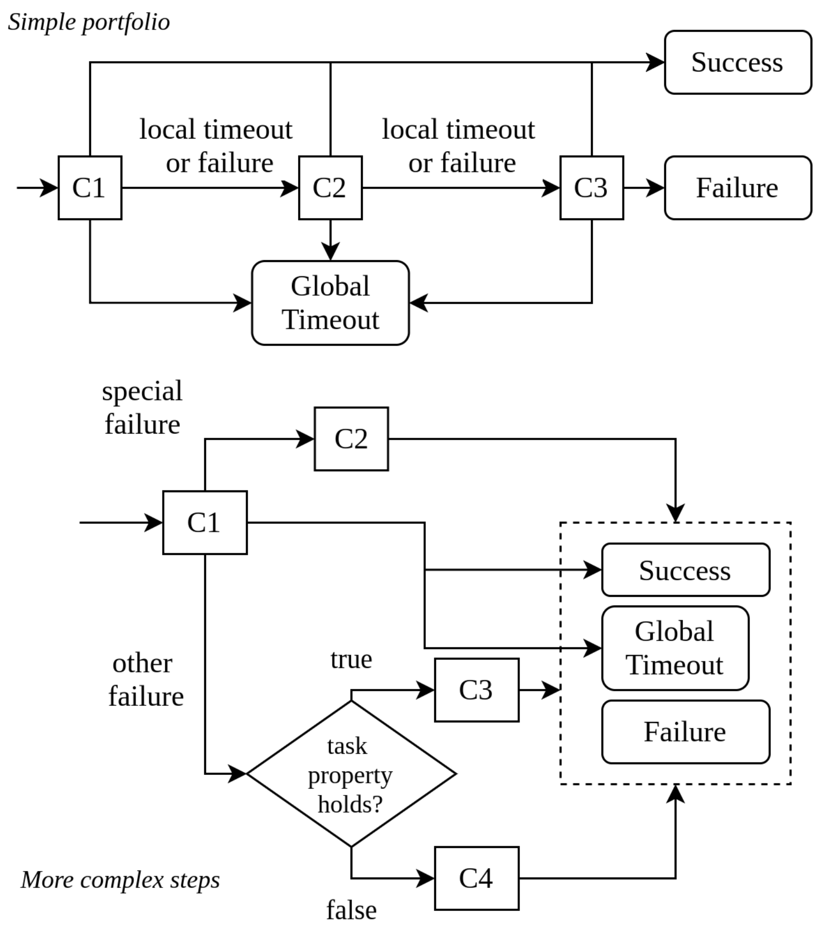


Figure 3.7: Process diagrams showing the earlier introduced simple sequential portfolio employing three configurations (C1,C2,C3) *(top)* and an example of a more complex assembly using algorithm selection and runtime information with four configurations (C1-C4) *(bottom)*.

have worked with another configuration better. But if the heuristics were well prepared, then such a portfolio might perform much better on diverse tasks both in time (e.g. trying less configurations before finding the right one) and in the number of solved tasks (e.g. more configurations can be utilized).

3.4.5 Choosing Configurations for Dynamic Configuration Selection

Working with a tool usually results in knowing the well-working configurations and algorithms that are “usually worth a try". Building on this knowledge is a possible way of assembling a portfolio and the broader this knowledge, the more general this portfolio might become.

A few points to consider when choosing configurations:

- Try to use a diverse set of configurations; if there are more *well-performing* configurations that work well on similar tasks, it might be better to employ only one of these.
- Bring the usually *fast-failing* configurations to the beginning – they won't waste much time in cases where they do not work.
- Use many *branches*; utilize the possible *runtime and algorithm selection methods* to avoid long sequential executions that slices the available time up into small intervals.
- Set local timeouts, where sensible:
  - if a configuration is prone to timeouts
  - if it is not the last configuration and the rest of the configurations have an actual role (i.e. not just placed at the end as “last resorts" to utilize the remaining time if the current configuration fails)

Inclusion and Exclusion It might also be worth to think about choosing configurations in terms of *inclusion* and *exclusion*, i.e. there might be some options that will throw an *error* for certain tasks, because they lack support for a certain language or model element and *excluding* the superfluous execution of that configuration would be beneficial. While other configurations should be *included* for certain tasks as they usually perform well in a given category or the presence of a given property.

Extra steps Also keep in mind the unique possibilities of the tool when planning the portfolio, e.g. if the tool is capable of giving a counterexample that is easy to validate (like an executable counterexample) then it might be beneficial to add a step of counterexample validation [1][8], which, if ends in a failure, lets the portfolio continue instead of giving a false success.

Separate Configuration Options Until this point I talked about configurations as *complete parametrizations* of the tool. If the tool has separate, mostly *independent configuration options* instead, the number of possible complete configurations can easily explode.

If the connections between the options are loose, i.e. it seems feasible to add a runtime decision only on a single or a few (but not all) configuration options then it might be worth to utilize this and make a series of decisions on single options until reaching a complete configuration to execute. But it is more likely that the options are bound more tightly *(e.g. if option A is set to true, the configuration will practically only work well if option B is also set to true)* and it is not worth it to take the practically unusable configurations into account *(e.g. when option A is true, but option B is false).*

Connecting Configurations to Properties Pairing certain property values (usually more than one) to a configuration that might perform well on them involves *experimenting* and *benchmarking* with several configurations on several *task sets.*

Both 1) having a hypothesis based on the algorithm and then verifying it with benchmarks or 2) finding a pattern based on the experiments and finding a logical explanation for it based on the algorithm and the tasks are valid solutions.

3.4.6 The Problem of Overfitting

One of the most discussed and highly-criticized issue of assembling portfolios is the lack of *transferability* and the problem of *overfitting* [47]. It is inevitable that the portfolio will have a scope – if no other than the application domain the assembler of the portfolio knows/experimented on. There will always be new input tasks that were not considered during portfolio assembly, but could be solved with a special configuration that was unused until that point – this is the issue of *limited transferability.* One approach to solve it might be the continuous expansion of the portfolio, adding more and more time and effort into it.

However *overfitting* can already happen during the initial assembly of the portfolio. When there is a really specialized group of inputs, it might be overfitting to create a rule based only on that group.

There is usually no clear line between well-working algorithm selection and overfitting. But when trying to prevent it, it is important

- to generalize how the tasks of this category are recognized (i.e. will there be any such programs besides this really special domain that might not come up ever again and if there is anything similar outside that domain, will we be able to recognize it or should these rules be loosened up a bit),
- and to find a logical explanation behind the decisions added to the portfolio (e.g. why is this working well in this case, how can it be explained based on the algorithm/configuration used). This might not always be possible, in that case try to experiment with the possibilities as exhaustively as possible.

3.5 Summary

To summarize, this chapter detailed a systematical collection of static and dynamic portfolio techniques for verification. Its structure is based on two diagrams detailing the possible portfolio features, which in themselves can serve as a checklist of ideas for any developer looking to design a portfolio for a tool.

Beside the already well-known algorithm selection problem and methods, the dynamic techniques also detailed novel possibilities on monitoring the analysis, intervening when needed in several ways and using the collected runtime knowledge when deciding on further configurations.

Furthermore a detailed example of portfolio creation, followed by advice on configuration collecting methods and warnings on overfitting were given in the end.

In the next chapter the methods of this chapter are put into practice by creating a dynamic portfolio for C software verification in the tool Theta.

Chapter 4

Designing a Dynamic Portfolio for Abstraction Refinement-Based Analysis

To evaluate the applicability and benefits of the methods of Chapter 3, a dynamic portfolio following the proposed guidelines has to be realized on a concrete tool.

Theta[1] is a highly configurable model checking framework using *abstraction refinement-based (CEGAR)* analysis (more on the tool in Section 2.4.2. But Theta was made mainly to benchmark, research and improve CEGAR (while supporting several formalisms and solvers).

One of the main advantages of Theta is that it is not exclusively a software verification tool, but is capable of handling other formalisms as well, such as formalisms for timed automata and state machines. For this reason, the created dynamic portfolio will be transferable to these formalisms in the future and can be compared to the results of this work. Furthermore Theta is open-source and it is easy to add new features due to its modular structure. It is also interesting to see how the tool being highly configurable, but using only a single "base" algorithm results in an unusually granular solution in the resulting portfolio.

In this chapter the Theta-specific verification process will be introduced (Section 4.1) and the empirical methods I used in this work are described (Section 4.2). Then I will introduce a modified CEGAR loop that improves the capabilities of the explicit analysis using runtime information (Section 4.3). I will then analyze the algorithm selection possibilities in the tool (Section 4.4) and last a dynamic portfolio is assembled by joining these methods together into a single dynamic portfolio for verifying C programs (Section 4.5).

4.1 The verification Process of Theta for C Programs

The Theta specific version of the general verification process introduced in Section Section 3.2 is shown in Figure 4.1. The model transformation uses an *ANTLR grammar* [44] to parse the input C program. During that process the tool builds a *CFA* to later use CEGAR on it.

[1]https://github.com/ftsrg/theta

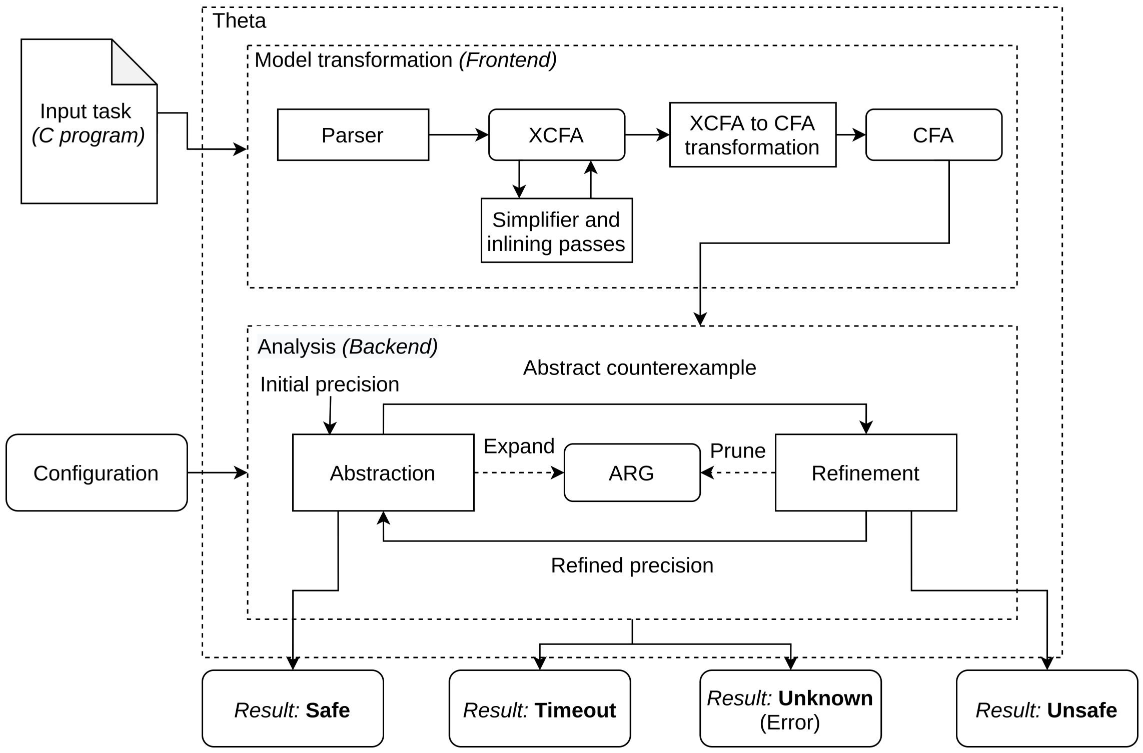


Figure 4.1: This figure is the Theta specific version of Figure 3.1, using the XCFA frontend, which is able to take C programs as input tasks.

The analysis then executes the CEGAR loop, containing the abstraction and refinement algorithms. When that stops, a result is given, which might be safe or unsafe (property holds or does not hold) in the successful case, or the tool might return some kind of error (e.g. a stack overflow or an SMT-solver exception), but it also possible that it finishes with a timeout after a while, if a time limit was given.

4.2 The Empirical Method of Designing the Portfolio

4.2.1 Manually Assembling a Portfolio

The portfolio was designed and assembled manually by carefully considering the advantages and disadvantages of Theta's configuration options.

The reason of not using *machine learning* is the limited number of *diverse tasks* that I could work and later benchmark on. Although the number of all available tasks in the benchmarking set is fairly large (there are more than 5000 tasks using the reachability property), the diversity of the tasks might not be really high – many of the tasks are artifically made or simplified and there are many variants, that are really similar.

Furthermore even though the need to use the configuration options of *Theta* more efficiently is high, the frontend of the tool is currently limited in support of C language elements (e.g. it cannot work with function pointers and has only limited support for structs and pointer arithmetics). In the end, there is simply not enough diverse tasks to use a neural network or anything similar on and get reliable results. However most of my work could be integrated with such techniques and as the limiting features of the verification tool are rapidly advancing, it might be feasible to add as future work.

4.2.2 Empirical Methods Used

The engineering knowledge to create this portfolio and every technique added to it was gained through a mix of theoretical knowledge, earlier experiences, small draft benchmarks and manual observation of the running analysis.

The earlier experiences include last year's work on preparing Theta to compete on the *International Competition on Software Verification (SV-COMP)* [9][1], albeit not with the frontend used in this work – it competed with the frontend Gazer[2] under the name *Gazer-Theta*. The change of frontend seems like a technical detail, but as Gazer is an LLVM-based[3] frontend, using *single static assignment variables* (SSA) and many black-box optimizations, it would be much harder to deduce connections between the input tasks and Theta's configurations and the list of supported and unsupported tasks changed a lot as well.

To prepare Theta and Gazer to the competition I executed systematic benchmarks on large sets of tasks of SV-COMP in several categories, which resulted in experience in several domains:

- Tool integration and usage of the benchmarking framework *Benchexec* [14]
- Structure of the task categories of *SV-COMP* and knowledge about the size and properties of the categories
- The relevant *configuration options* of Theta that usually matter the most[4]

Although Theta has changed a lot since last year (mainly in a technical and implementation sense), most of the above mentioned experiences are still valid and offer a base to start from.

The other source of experimental knowledge in this work was an *iterative set of small measurements* on different aspects of the tool on the SV-COMP tasks to discover properties of the tool and its configurations that can then be explained through the models and the CEGAR algorithm, but would not be trivial to find out without benchmarks (e.g. that explicit analysis is prone to get stuck, see Section 4.3).

To explain and reason about the above mentioned properties, beside applying *theoretical knowledge* I mostly focused on *empirical methods* observing the tools operation while executing the analysis using debugger and profiling tools.

4.2.3 High-level Design Decisions

The tool Theta already had a sequential portfolio[1], therefore my goal was to design and implement a more complex, dynamic portfolio.

Figure 4.2 illustrates the features selected and implemented in this new portfolio. I concentrated on the following aspects, that could potentially improve the performance of Theta based on the previous experiences.

- Monitoring and intervention

[2]Gazer tool `https://github.com/ftsrg/gazer`

[3]LLVM project: `https://llvm.org/`

[4]Documentation of the tool on the options: `https://github.com/ftsrg/theta/blob/master/doc/CEGAR-algorithms.md`

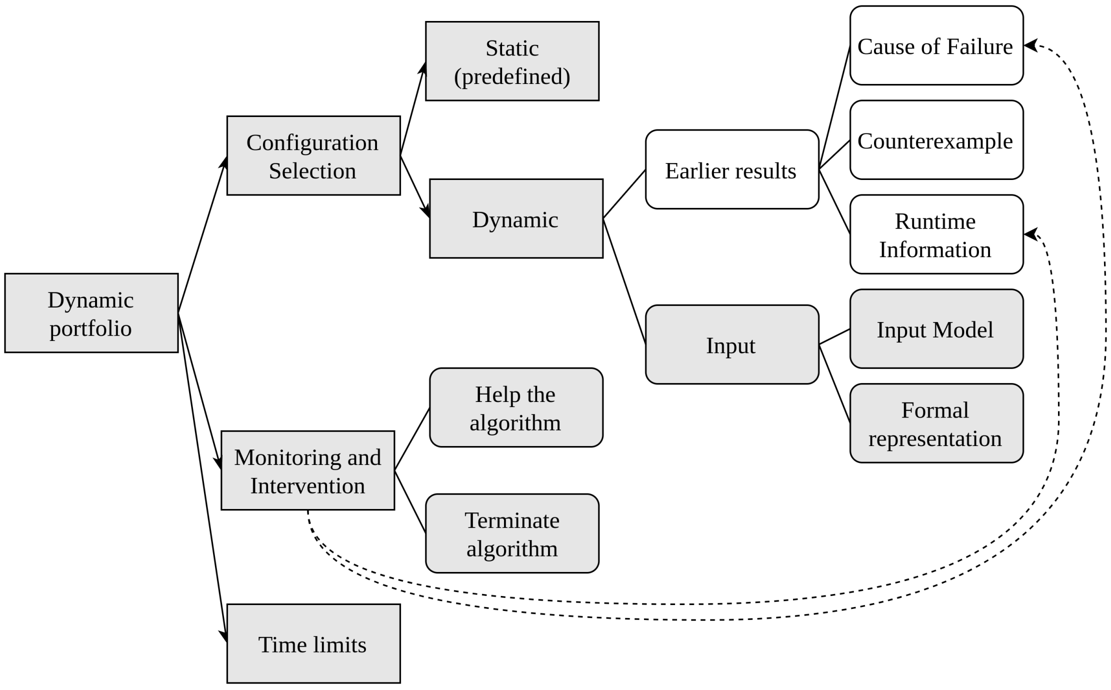


Figure 4.2: The diagram of dynamic portfolio feature (Figure 3.5), but the features that were added in this chapter's dynamic portfolio are colored grey.

- Configuration selection based on the input program

The following sections will introduce the details for these improvements.

4.3 An Improved CEGAR Algorithm Using Runtime Information

4.3.1 Typical Issues during Verification

As we can see in Figure 4.1, in the case of successful verification Theta outputs a result, namely if the input program was *safe* or *unsafe*. But verification can also be unsuccessful. Either we run out of time and patience and the result is a *timeout*, or we run out of some other resource (e.g. memory or the stack) and the result is *unknown*. Technical issues, such as some kind of unsupported C construct, SMT solver issues or bugs will also result in *unknown*. There can be many different *unknown* results, which can mostly be solved through implementing new features and fixes in the verification tool.

But what will be of greater interest for us in the unsuccessful cases are the *timeouts*. Although at first it may seem like, that such an output simply implies performance problems, there are more than one typical pattern here worth observing.

Timeouts due to performance problems As we have seen, the CEGAR loop consists mainly of two algorithms: *abstraction* and *refinement*. In problematic cases either can be quite slow. Abstraction takes a long time if the ARG being built is really large and the expanded nodes constantly result in many new nodes to be expanded – practically resulting in a timeout in many cases, depending on the available resources.

Refinement, on the other hand, is based on *SMT solvers*, which can also require a huge amount of time to solve the problem given to them. In many cases we have observed timeouts after spending a long time in a given iteration in either of these algorithms. In special cases verification can't even build the first iteration of the ARG in time.

These are special cases of performance problems and tackling them is the same as *improving the performance of certain module in Theta* through adding new and optimizing existing features.

There exists of course the more general case, where the size and complexity of the input task is simply so high that neither the abstraction nor the refinement algorithm can tackle it in any sensible time – that should also get better with the tool's general progress, but it is probably the hardest issue to address from the above.

Timeouts due to algorithmic issues Most abstract domains are not complete – in many cases they can only be refined up to a certain point and even in their most refined state they do not become identical with the concrete state space. The straightforward example is the *explicit* domain – while $x = 0$ can be formulated and tracked, *"x can be anything but 0"* cannot. Furthermore, even *Cartesian predicate abstraction* is incomplete, as it cannot express XOR connections, although in practice this comes up much less regularly.

Due to this incompleteness, an unexpected type of timeouts come up fairly regularly, namely that the algorithm gets into an infinite loop of the same few iterations. It builds the same *ARGs* over and over again without being able to improve precision and it finds and checks the same *abstract counterexample* over and over again. This rarely happens in the *predicate abstraction domain*, but comes up often in the *explicit domain.*

Figure 4.3 shows a simple example of this with a single ARG being built over and over again. Although the variable x is present in the precision, it might happen, that the value of x is unknown. On the other hand, the path leading to *locErr* through *locX* and *locY* is infeasible, if there was an earlier predicate, which makes $x > 0$ false (e.g. $x <= 0$), but the explicit domain cannot express and store this predicate, thus x remains unknown at this point. But if the ARG is pruned back to the same spot and the precision also remains the same, the algorithm will build the ARG over and over again in the direction of the *infeasible counterexample*, thus getting *stuck.* Although here it is only a single ARG that the analysis iterates infinitely on, but in many cases it is a sequence of ARGs that forms a loop of infinite iterations.

The explicit domain, although limited in expressiveness, is generally much faster than any predicate abstraction of the tool. But that is hard to utilize efficiently if it regularly gets stuck on other tasks. Contrary to the earlier examples of timeouts, where there is an actual performance problem, here it is caused by the lack of such information that the explicit domain could use. The big difference between these is that it can be monitored if the algorithm makes no refinement progress (it is "stuck") at a certain point of time or not. In the following subsection a run-time algorithmic improvement fulfilling this task is introduced.

4.3.2 Monitoring ARGs and Counterexamples

The goal of my work in this section is to detect when there is no refinement progress due to incomplete domains and mitigate or stop the resulting infinite loop in the algorithm.

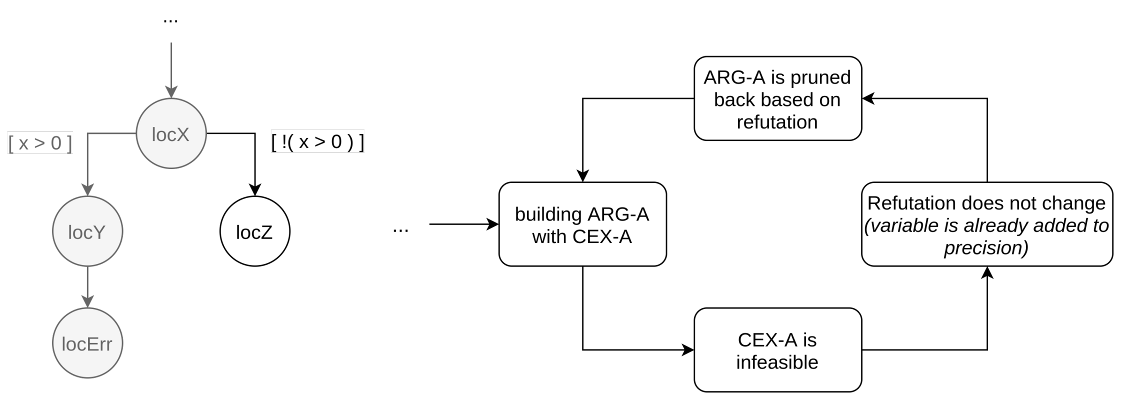

...
locX
[x > 0]
[!(x > 0)]
locY
locZ
locErr
...
building ARG-A with CEX-A
CEX-A is infeasible
Refutation does not change (variable is already added to precision)
ARG-A is pruned back based on refutation

Figure 4.3: Part of a CFA and a process diagram showing how the explicit analysis can get stuck in an infinite loop.

The algorithmic improvements are formalized and added to the already implemented abstraction and refinement algorithms in Algorithm 4.1 and 4.2 *(the newly added parts are highlighted)*.

Detecting infinite loops To detect when the algorithm is "stuck", we have to detect if the ARG and precision changes throughout the iterations of CEGAR and for this we will need to store the ARGs and precisions of earlier iterations. Based on Figure 4.3 it might seem like that comparing only the last two ARGs and precisions might be enough, but it is possible that a set of ARGs form the infinite loop instead of just one, thus comparing only the last ARGs is not sufficient.

Thus a set collecting *abstract reachability graph* and *precision* pairs is created and continuously expanded with new ARGs – the ARG that is passed from the abstractor to the refiner is added to this set in each iteration. Before it is added, the set of ARGs is checked for containment of the new ARG and if it is already present then we know, that the analysis is stuck.

The reason why precisions are also added instead of only collecting the ARGs is that the change of precision means progress in the algorithm and it might seem rare in the explicit domain, but none the less it is possible to build the same ARG with different precisions. This occurs more frequently in predicate-based domains.

Mitigation of infinite loops When the algorithm makes no refinement progress, it is a straightforward decision to *stop* it, but there is another possible opportunity as well.

Most CEGAR tools employ a lazy strategy, i.e. they build the ARG only until the first counterexample is found and then the abstraction stops and refinement starts on that counterexample immediately. When there is no refinement progress in the algorithm then this same first counterexample is found again and again in each iteration – but if the abstractor would build the ARG further instead, it might find other counterexamples, with which the refinement could progress further.

This possible issue with the lazy strategy is demonstrated in Figure 4.4 on a *partially expanded ARG* of an *explicit analysis*. The grey loc_6 was not discovered, as the abstractor expanded loc_3 and its successors instead and then proceeded to refine the counterexample on that path instead of building the ARG further. But the counterexample is *not feasible*

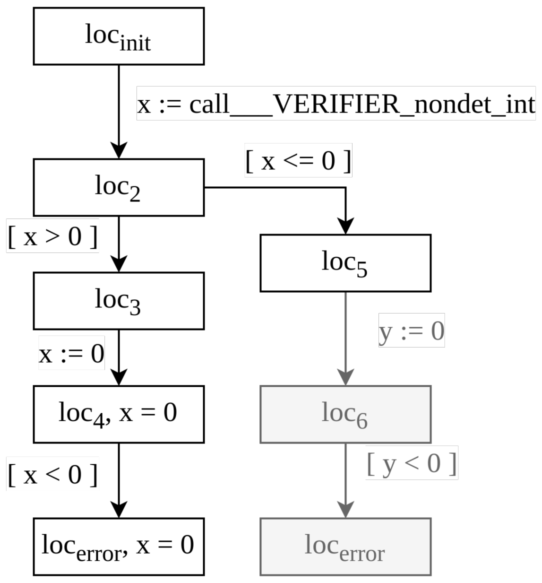


Figure 4.4: A partial ARG as an example of the lazy strategy. The explicit analysis tracks x only.

and there are *no new variables* in this path to be added to the precision, as x is tracked already. Thus there is no refinement progress and the ARG will be pruned and then built back again to this same structure. If the grey counterexample through loc_6 could also be discovered and refined instead, then y would be added to the precision and the algorithm could progress further.

To mitigate such cases as in Figure 4.4, the counterexamples have to be collected into a set after refinement. This set then can be used to check if a newly-found counterexample was refined earlier or not. The collected counterexamples are all infeasible abstract counterexamples, as the algorithm would have stopped already if there was a feasible counterexample found.

When a counterexample reappears while the abstractor is expanding the ARG, the abstraction is not stopped, instead the ARG is built further until a new counterexample is found. The refiner will need to choose this new counterexample for refinement instead of the first one. If there is no new counterexample to be found, it is concluded that the analysis is stuck and cannot be saved and thus it is stopped with an inconclusive result.

4.3.3 Practical Considerations

Storage Overhead Storing a large quantity of abstract reachability graphs would require a tremendous amount of memory, so to avoid this, both the ARG, precision pairs and the counterexamples are stored and compared only as hash codes.

Using hash codes carries a slight chance of hash collisions. In case of a hash collision, two *counterexamples* or two *ARG precision pairs* might be stored with the same hash codes, causing the checks for equality to give a false positive result of the checked constructs being equal, even though they are not the same.

If the colliding hash codes are of ARG precision pairs, then the algorithm might build the ARG further, if the counterexample was stored earlier already, which could rarely be the case. That would mean a missed opportunity for refinement with that ARG and

Algorithm 4.1: 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
         $ArgSet :< arg, \pi >$: the pair <arg,π> is an abstract reachability graph
and a precision, that occurred together earlier at least once
         $CexSet$: A set of counterexamples that was already found and refined
earlier.
output: (safe or unsafe, $ARG$, $ArgSet, CexSet$)
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 and the counterexample is new with respect to the
       current ARG and precision
5      $\sigma = ((l_1, s_1), op_1, \ldots, op_{n-1}, (l_n, s_n))$ := path to unsafe node (with $l_E$) from
       $ARG$
6      if $l = l_E$ and not ($ARG \in ArgSet$ and $\sigma \in CexSet$) then
7          $ArgSet := ArgSet \cup ARG$
8          return (unsafe, $ARG$, $ArgSet, CexSet$)
9      // Check if $(l, s)$ can be covered
10     else if $\exists (l', s') \in N : (l, s) \sqsubseteq_L (l', s')$ then
11         $C := C \cup \{(l, s, l', s')\}$ // Add covered-by edge
12     // Otherwise $(l, s)$ gets expanded
13     else
14         foreach $(l', s') \in T_L((l, s), \pi_L) \setminus \bot_L$ do
15             waitlist := waitlist $\cup \{(l', s')\}$
16             $N := N \cup \{(l', s')\}$ // Add new node
17             $E := E \cup \{(l, s, op, l', s')\}$ // Add successor edge
18 if $\exists (l_E, s) \in N$ then
19     // There are no new counterexamples to be found in the ARG, so the analysis
       has to stop with an inconclusive result
20     return (inconclusive, $ARG$, $ArgSet, CexSet$)
21 else
22     return (safe, $ARG$, $ArgSet, CexSet$)
```

counterexample and could theoretically cause an inconclusive result, but it cannot cause a false one.

On the other hand, if counterexamples' hash codes collide, then the algorithm might miss the refinement of a counterexample – this cannot lead to a false result, but it might lead to an inconclusive result. For that to happen, the evaded counterexample should have been crucial to get a result and a similar, but more refined counterexample could not be created either. It is easy to see, that the chance for this is probably negligible.

Altering to Refinement with Multiple Counterexamples The introduced technique depends on the fact that the abstractor normally stops as soon as it finds the first abstract counterexample. Theta already has the refinement possibility *MULTI_SEQ*,

Algorithm 4.2: Refinement algorithm.

```
input  : $ARG = (N, E, C)$: unsafe abstract reachability graph
         $l_E$: error location
         $\pi_L$: current precision
         $CexSet$: A set of counterexamples that was already found and refined
 earlier.
output: (unsafe or spurious, $\pi'_L$, $ARG$)
1  $\sigma = ((l_1, s_1), op_1, \dots, op_{n-1}, (l_n, s_n)) :=$ path to unsafe node (with $l_E$) from $ARG$
2  // Feasibility check
3  if $s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge \dots \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$ is satisfiable then return (unsafe, $\pi_L$, $ARG$)
4  else
5      // Record the new counterexample to CexSet
6      $CexSet := CexSet \cup \sigma$
7      $(I_1, \dots, I_n) :=$ get interpolant for $\sigma$
8      // Precision adjustment
9      $(\pi_1, \dots, \pi_n) :=$ map interpolant $(I_1, \dots, I_n)$ to precisions
10     $\pi'_L := \pi_L$
11     if $\pi_L$ is local then $\pi'_L(l_i) := \pi'_L(l_i) \cup \pi_i$ for each $l_i$ in $\sigma$
12     else $\pi'_L(l) := \pi'_L(l) \cup \bigcup_{1 \le i \le n} \pi_i$ for each $l \in L$
13     // Pruning
14     $i :=$ lowest index for which $I_i \notin \{true, false\}$
15     $N_i :=$ all nodes in the subtree rooted at $(l_i, s_i)$
16     $N := N \setminus N_i$ // Prune nodes
17     $E := \{(n_1, op, n_2) \in E \mid n_1 \notin N_i \wedge n_2 \notin N_i\}$ // Prune successor edges
18     $C := \{(n_1, n_2) \in C \mid n_1 \notin N_i \wedge n_2 \notin N_i\}$ // Prune covered-by edges
19     return (spurious, $\pi'_L$, $ARG$)
```

which exhaustively builds the *ARG* until it cannot be expanded further and *all counterexamples* are found (and then uses all the counterexamples for refinement).

In that case, there is no sense in trying to find more counterexamples if there aren't any new ones so the algorithm is stopped instantly when an *ARG-precision pair* comes up a second time. So when using the *MULTI_SEQ* refinement, a variant of the modification is used, which does not store the counterexamples and cannot mitigate the infinite loops in any ways, but is still capable of stopping the analysis, when needed.

4.4 Configuration Selection Possibilities

In this section I discuss the capabilities of the different *abstract domains*, *refinements* and *arithmetics* implemented in Theta and how the performance of these can be connected to certain input task properties.

4.4.1 Choosing Abstract Domains

Precision - Efficiency Trade-off Generally the most influential configuration option of a CEGAR analysis is the abstract domain. The abstract domain determines what information can be extracted on the possible variable values.

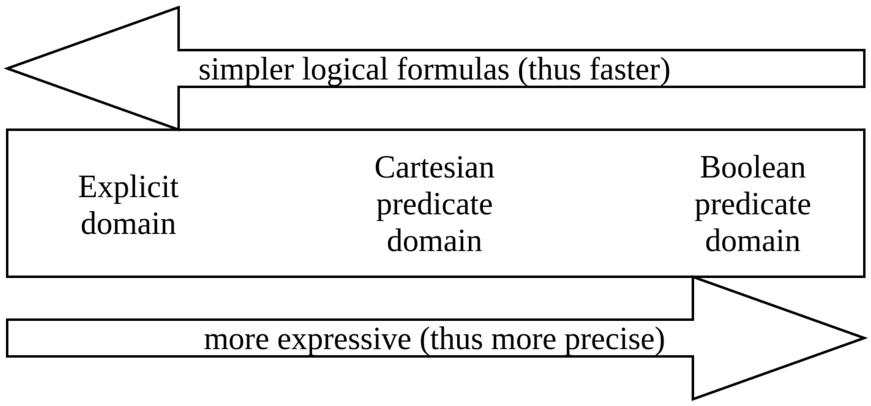

simpler logical formulas (thus faster)
Explicit domain
Cartesian predicate domain
Boolean predicate domain
more expressive (thus more precise)

Figure 4.5: The trade-off between efficiency and precision in terms of abstract domains.

Typically choosing a domain is a trade-off between efficiency and precision as shown in Figure 4.5. The explicit domain can only express a finite number of equalities over variables with constants, limiting expressiveness; but due to this, the logical formulas given to the SMT solver are simple and thus this domain is generally faster than the others. Cartesian predicate domain keeps track of conjunction of logical predicates, while Boolean does this with arbitrary Boolean combinations instead of conjuncts, which makes calculations more expensive [5].

The earlier introduced *modified CEGAR with runtime monitoring* (Section 4.3) pairs well with the *precision-efficiency trade-off* of explicit and predicate domains. If a program does not contain enough information for the explicit analysis' success, it will eventually get stuck. At that point, it can be stopped and a predicate analysis can be started instead.

The Strengths of Predicate Analysis Explicit abstraction cannot express value intervals (only concrete values), while predicate analysis can. In practice, the typical construct that regularly turns out to be connected to value intervals are *loops*. Let us assume that there is a loop with a *counter variable*. Explicit abstraction will typically *unfold* in the form of many abstract states while following the value of the counter. Predicate abstraction on the other hand might be able to find a predicate that *covers* multiple similar iterations of the loop and can use that to cut down on the number of abstract states (e.g. a loop with the condition $x > 1000$ can result in at least a 1000 states only because of x, even though with predicate abstraction $x > 1000$ could be enough). In such cases this can easily make the predicate domain a winner over explicit analysis. However, a fairly low *cyclomatic complexity* is also required, as otherwise the task is probably too large to be handled by the expressive, but slow predicate abstraction.

Thus, the heuristic proposed is: predicate analysis should be executed, if

- there is at least one *while loop* in the program and
- the task has a *cyclomatic complexity* below *30*, a value based on earlier observations of results and tasks

Furthermore, the time limit of the predicate analysis will be set to be fairly short, as if the predicates do not work well in a small amount of time then the task might be difficult to solve with predicates, but the reason for this might lie in other program attributes instead of cyclomatic complexity.

Future Improvement While talking about abstraction, it also has to be mentioned that there are several more granular solutions for choosing the abstract domain per variables based on variable roles or other runtime properties [3]. These might offer a better solution than what we currently have in the tool and it is under active research to develop and implement such techniques in Theta, but for now product abstraction only works with another formalism[5]. As future work we would like to benchmark and add these to this portfolio as well when it will be available.

4.4.2 Choosing Refinements

If the abstraction algorithm's most influential configuration option is the abstract domain, then the most influential option for the refinement algorithm must be the refinement strategy. What we have seen here generally is that sequence interpolants work well in the explicit domain and binary backwards interpolation works well with predicate abstraction while the others fall behind these in most cases.

Precisions with Faster Convergence In special cases, the refinement *UNSAT_CORE* works well, which instead of using an interpolant uses an *unsat core* [41] of the logical formulas to add new variables to the precision (it works only in the *explicit domain*). This special case includes programs, where a larger group of variables (preferably with known values) are important for success, but the task is also large. The explanation for that is that an unsat core can expand the precision in each iteration with several variables instead of just one or just a few, while interpolants will not do that as they localize to the point of the counterexample, where there is a contradiction making the path infeasible.

A configuration with a more brute-force solution, but similar strengths to the *UNSAT_CORE* refinement is using the *ALLVARS* initial precision (instead of the default *EMPTY* initial precision). This adds every variable to the precision initially, practically maximizing it. From then on, the analysis becomes similar to *bounded model checking* in the sense, that the precision remains the same and iterations are simply searches for new abstract counterexamples, which are then checked by the refiner.

The weakness of both is the possibility of building really large *ARGs* due to the large number of variables included in the precision. In other words, the abstraction might be too fine, causing performance issues.

Tasks with Many Variables to Track The above mentioned configuration qualities were first observed on *SV-COMP* programs encoding *event-condition-action (ECA) systems* [38], where there is only a single input and a single output variable, but many variables with known values that are used extensively in branch conditions. These tasks have many common features:

- there is a single input (with a non-deterministic value) and a single output variable

[5]See product abstraction domains here: https://github.com/ftsrg/theta/blob/master/doc/CEGAR-algorithms.md

- the point of the program is to calculate the output variable
- the structure of the program is similar to some kind of encoded state machine:
  - there is a single outer *while(true)*, which we break out of, when the output is calculated
  - they have many variables with an initially assigned value and these are used in branch conditions
  - there are many branches inside the calculation with conjunctions of equality checks between variables and constants
- these systems can easily grow quite large by adding more and more steps to calculating the output, resulting a task so large, that after a certain point Theta won't be able to solve it in a reasonable time

Generalization of the Observations Based on the above two paragraphs I created a more general algorithm selection heuristic, as it might be possible that there are non-ECA tasks, where these configurations could also improve efficiency. The key to generalization lies in the variables of the tasks.

There is a good chance, that using an *ALLVARS* initial precision or the *UNSAT_CORE* refinement with its fast growing precision is a good choice, if

- there is a "handful" of *known* and only a few *unknown* variables
- and the known variables' values are worth to follow, because they play an active role in the *feasibility of abstract counterexamples*

In the tasks that we are currently observing, most variables with known constant values are probably used to direct the program flow (e.g. in practice a hardcoded filename would be a variable with a constant value which will not be used in any conditions, but such technical values are atypical in the tasks that Theta currently supports).

Based on that, two simple metrics, namely the number of variables and the number of variables with non-deterministic value (e.g. input variables) are enough to decide if these configurations are worth to try. These task properties can be easily extracted from the *Control Flow Automaton*, as it directly stores the variables and they receive their *non-deterministic values* on *CFA* edges labeled with the *havoc* operation.

It is clear that this heuristic might need to be developed further in the future as the number of tasks that the tool can solve rises, especially in the case of more realistic programs. But it gives a good foundation to start off of.

4.4.3 Choosing Arithmetics

Some SMT solvers can handle different arithmetics and Z3 [28], the solver used by Theta, is one of these. Theta also has a type system, which can express either integer or bitvector arithmetics in the *Control Flow Automaton.* Generally *integer arithmetics* are much faster, as they do not have to handle each bit separately, while *bitvector arithmetics* are able to handle bitwise operations and floating point values on a bit-precise level.

From the algorithmic selection viewpoint it is clear, that we should prefer integer arithmetics and use bitvector arithmetics only, when it is needed (e.g. when floats are present in the input task). The earlier sections on configuration were about finding and choosing the *preferable* configuration, that might perform better – but the rest weren't completely incapable either. Opposite to that, here the integer arithmetic will have to be *excluded* based on the given task.

Most refinement options in Theta cannot handle bitvector arithmetics. The ones that can are still in an experimental phase. From these experimental refinements, *NWT_IT_WP* [31] was observed to have the least performance issues. However it is still generally slow, so it will be paired with the explicit domain to try and counteract the issues with its speed.

The *"best-effort"* strategy to be used here is the following:

- First we start with the explicit domain, without a time limit
- If that gets stuck, we switch the domain to predicate analysis as a last resort

Even with that, using bitvector arithmetics will most likely result in many *timeouts*. However that is a performance problem that will only be solved with the further development of these features.

4.4.4 Other Options

Theta currently has around ten possible *input flags* with many different values. These configuration options were all considered to be used in this work, but in the end they proved to be difficult to use in an algorithm selection setting.

However any configuration option of the tool might be hiding such strengths in special cases as the ones that were introduced above. But without input tasks where these strengths appear, it is not possible to find, utilize and evaluate these. So as the tool and its frontend supports more and more diverse tasks, these techniques should be continually expanded with new additions.

It is also worth to mention that such options without the required domain expertise could also be investigated through machine learning techniques to help discover missing connections between input properties and the options in the future.

4.5 Assembling the Complex Dynamic Portfolio

In Section 4.3 and 4.4 several techniques were realized based on the methods introduced in Chapter 3. The next step is to put these together into a portfolio in a way that it utilizes the tool as efficiently as possible.

4.5.1 Portfolio Introduction

The assembled dynamic portfolio is visualised in Figure 4.6. Each rectangle stands for a *single configuration* and if any one of these succeeds, the portfolio also stops with a *success*. Otherwise it starts the next configuration and it does this until it runs out of configurations or the global 900 seconds *time limit*.

The Bitvector Arithmetic Path First, the *arithmetic* to be used is chosen – a simple check was implemented for such language elements that cannot be used with integer arithmetics (i.e. floating point types and bitwise operations). If some of these are present then bitvector arithmetics are required and the *NWT_IT_WP* refinement is used, first with the *explicit domain* and if that fails then with the *Cartesian predicate domain*, just as described in Section 4.4.3.

The Integer Arithmetic Path On the other branch, we apply the short runs if the input is fairly simple, but has loops, as introduced in Section 4.4.1. If this was skipped or unsuccessful, an explicit domain configuration is next.

Just as presented in Section 4.4.2, the *number of variables* and the *number of variables with non-deterministic values* are used to decide on the initial precision here (i.e. whether we should use the *ALLVARS* option). Although *ALLVARS* seems to be the quicker option for now, it would also be possible to use an empty initial precision with the *UNSAT_CORE* refinement instead – but this can easily change in the future, when more tasks are available to experiment on.

The *explicit analysis* is set to have a fairly long time, so it has a chance on more complex tasks as well and it is prone to get stuck and thus get stopped in the difficult cases anyways. In any type of *unsuccessful case* a generally well performing *Cartesian predicate analysis* is started in the remaining time. In the rare case of this ending early in some kind of technical error, we try a *Boolean predicate analysis* as a backup.

4.5.2 Technical Details

The inner timeouts are not absolute seconds in reality, rather fractions of the remaining time after parsing (e.g. if parsing the task requires 30 seconds, the timeout for the explicit analysis will be started with a timeout of $500 \cdot 850/900$ seconds).

It is an important feature of this portfolio that it is directly implemented in the verification tool itself so that model transformation happens only once and after that every analysis works on the same Control Flow Automaton.

4.5.3 Summarizing the Completed Portfolio

Combining all the designed techniques above, the assembled *dynamic portfolio* utilizes both *algorithm selection* and *runtime monitoring and intervention techniques*, which complement each other. It gives a "one-click" option for verification instead of having to manually configure the tool. It uses information extracted both from the original *input program* (while loops, floating point types and bitwise operations) and from the *formal representation* of that program (cyclomatic complexity, number of all variables/variables with non-deterministic values). It is optimized for a reasonable time (15 minutes) with added *local timeouts*. Although earlier results are currently unused, a counterexample check will be definitiely added in the future.

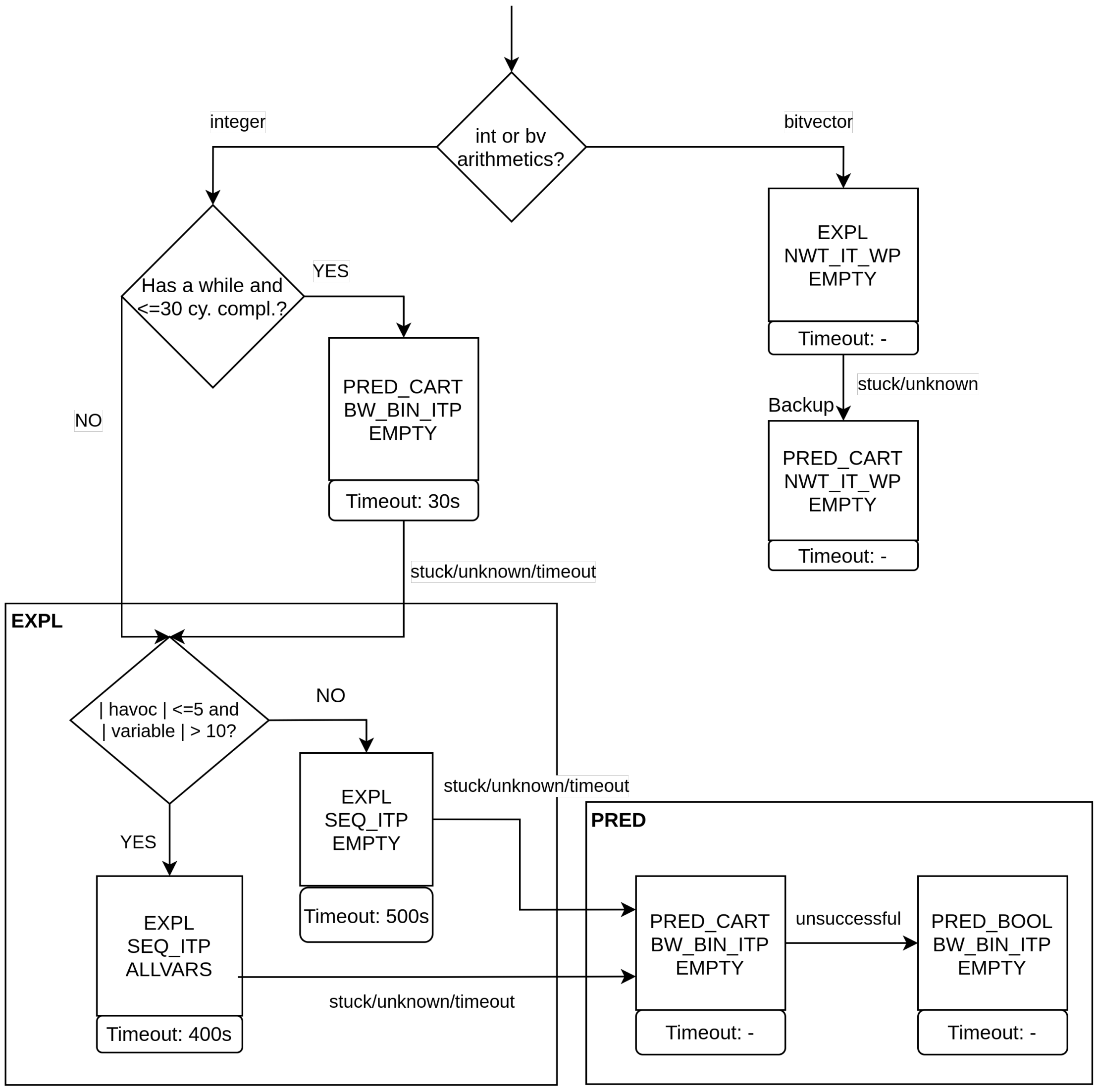


Figure 4.6: Process model of the assembled portfolio

Chapter 5

Evaluation

This chapter reports the evaluation of the *complex portfolio* designed in Chapter 4 through executing it on a large number of C programs. It starts with the experiment design of the benchmark, detailing the *input tasks*, *hardware*, *benchmarking framework*, *baseline configurations* and *research questions* in Section 5.1. The research questions are answered in Section 5.2 based on the results of the experiments and then further discussion is given on the experiment in Section 5.3.

5.1 Experiment Design

In the following section the design choices of the evaluation benchmark are introduced. These include the formulated research questions (Section 5.1.1, the verification input tasks used (Section 5.1.2); the subjects, the structure of the baseline configurations (Section 5.1.3), technical information about the execution environment (Section 5.1.5) and the variables of the experiment (Section 5.1.4).

The configuration files required to reproduce the experiment and the data of the results used in this evaluation are available under the DOI 10.5281/zenodo.5605708[1].

5.1.1 Research Questions

The following questions will help to evaluate how much the complex portfolio can help the performance of Theta on C programs. The two main goals of creating a portfolio was to enable the tool to solve more input tasks and to do this in less time, if possible. The questions reflect these goals.

RQ1 How does the complex portfolio perform in number of solved tasks compared to the baseline configurations?

RQ2 How does the complex portfolio perform in terms of CPU time compared to the baseline configurations?

[1]`https://doi.org/10.5281/zenodo.5605708`

ControlFlow	Programs for which the correctness depends mostly on integers and control flow structure
Bitvectors	Programs, in which treatment of bit-operations are necessary
ECA	Programs containing event-condition-action systems
XCSP	Programs generated from constraint network XMLs
Sequentialized	Sequentialized concurrent SystemC programs
Loops	Tasks focused on loops

Figure 5.1: The evaluation will utilize tasks from the above sub-categories of Reach-Safety

5.1.2 Verification Tasks

The de-facto standard way when benchmarking C or Java verification tools is to use the benchmarking tasks of the International Competition on Software Verification (SV-COMP)[2]. This evaluation uses a set of *1250* of tasks chosen from the *ReachSafety category* of SV-COMP, as this category expects analysis of the reachability properties supported by Theta.

Overall the ReachSafety category contains almost *5000* tasks. Filtering these tasks was necessary to reduce them to a feasible amount to benchmark on and it was also useful because most tasks left out are impossible to solve by the current version of the tool (e.g. due to the frontend not supporting certain elements of C yet or due to the large size of a program making the model transformation impossible within a reasonable time limit) and it is a waste of time to execute several configurations on them.

Tasks were selected from the sub-categories *ControlFlow*, *BitVectors*, *ECA (event-condition-action systems)*, *Sequentialized*, *Loops* and *XCSP*. In Loops and Sequentialized a frontend-only run was also executed to filter out the tasks that fail in the frontend already.

5.1.3 Subject and Baseline Configurations

The main subject of this evaluation is the complex portfolio of Chapter 4, implemented in the verification tool Theta. The portfolio's performance is compared to three other configurations, two of which are just single CEGAR configurations (Figure 5.1.3) that perform well generally and the third is a sequential portfolio – a realization of the simple portfolio introduced in Section 3.3.

Single Baseline Configurations The configuration *expl* is known as a generally fast configuration, when it works, but otherwise it is prone to timeouts – not just because of high complexity tasks, but also because it can get stuck in the same iterations of the CEGAR loop (see Section 4.3). Opposed to this, *pred* is much more expressive due to using predicates, but also generally slower because of the overhead this expressiveness adds and thus it is more sensitive to the complexity or size of the input tasks (see Section 4.4.1).

Baseline Sequential Portfolio Both of these will be executed on all the tasks by themselves, but they are also the first two configurations of the third baseline, the *sequential-portfolio*. The third configuration in this portfolio is *newton-expl* (also included in Fig-

[2]https://sv-comp.sosy-lab.org/2022/

Single Configuration	**CEGAR options used**
expl	–domain EXPL –initprec EMPTY –search ERR –encoding LBE –refinement SEQ_ITP –maxenum 1 –precgranularity GLOBAL –prunestrategy LAZY
pred	–domain PRED_CART –initprec EMPTY –search ERR –encoding LBE –refinement BW_BIN_ITP –predsplit WHOLE –precgranularity GLOBAL –prunestrategy LAZY
newton-expl	–domain EXPL –initprec EMPTY –search ERR –encoding LBE –refinement NWT_IT_WP –maxenum 1 –precgranularity GLOBAL –prunestrategy LAZY

Figure 5.2: Theta configurations used in the baseline benchmarks. The first two (*expl, pred*) will be executed both as a single configuration and is also added to the sequential portfolio; *newton* will not be executed by itself, but will be used as the third, bitvector specialized configuration of the sequential portfolio.

ure 5.1.3), which uses the refinement *NWT_IT_WP*[3], an experimental feature capable of handling bitvector arithmetic, where needed. Each of these are assigned a *300 second CPU time limit* and if they fail earlier, the remaining time is equally divided between the remaining configurations, just as in the portfolio introduced in Section 3.3.

5.1.4 Variables

Category	Name	Type	Possible values
Input task	task task category	String Enum	YAML file name ControlFlow, ECA, Loops, etc.
Tool	configuration	Enum	complex-portfolio, sequential-portfolio, expl, pred
Metrics	status cputime memory	Enum Floating point number Floating point number	true, false, different error codes, TIMEOUT, OUT OF MEMORY, unknown) CPU time used during execution (in seconds) Peak memory usage (in bytes)

Figure 5.3: The variables of the experiment

The variables of the experiment are shown in Table 5.3.

Input task Each of the input tasks is a C program in a single .c or a preprocessed .i file. Beside each of these files there is a YAML file, used by benchexec and named after the C program, but with a .yml extension – these contain the correct result (whether the reachability property holds or not) and the name of the .c or .i file. The name of this file (mostly with the addition of the parent directory) identifies the input program. The programs are also grouped into disjunctive categories, detailed in Section 5.1.2.

[3]Documentation on all the options used in the configurations: https://github.com/ftsrg/theta/blob/master/doc/CEGAR-algorithms.md

Tool The category Tool consists of the configuration variable. In Section 5.1.3 the baseline and subject configurations were introduced already. Two of these are simply executions of single configurations (*expl, pred*) and the other two are portfolios, containing multiple configurations (*sequential-portfolio, complex-portfolio*).

Metrics The metrics we are most interested in are the number of successful verifications, encoded in *status* among other possible outcomes and the *CPU time* required by the analysis. Although it will not be a main focus, *memory usage* is also measured by Benchexec.

5.1.5 Execution Environment

The measurements were carried out on virtual machines, each with two dedicated cores (four logical) of an *AMD Ryzen 9 3900X 12-Core* processor and *10GB of memory.* The machines are running *Ubuntu 20.04.3 LTS* and the Java version used is *openjdk 11.0.11.* The global CPU time limit was set to *15 minutes* in the benchmarking framework, similarly to SV-COMP.

To execute the necessary benchmarks, the version 3.8 of the *Benchexec* benchmarking framework [14] was used, which, among others, is the framework used on SV-COMP and several other competitions. The results were collected to csv files by the framework, which was used to evaluate the benchmarks.

The benchmarks were executed with a version of Theta on the branch *xcfa-algorithm-selection*[4]. The tasks used were downloaded from the now archived version of the sv-benchmarks repository[5]

5.2 Results

In this section the above formulated questions are answered based on the results of running the benchmarks. The summarized data is visualized on heatmaps and column charts to compare the performance of the different configurations.

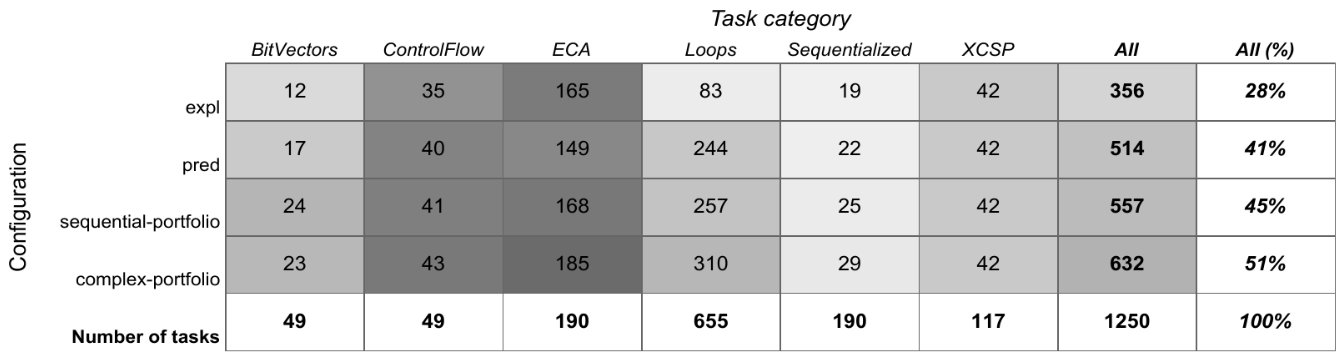
5.2.1 RQ1: Solved Tasks

How does the complex portfolio perform in number of solved tasks compared to the baseline configurations? The heatmap in Figure 5.4 shows the number of successfully solved tasks by category for each configuration. The complex portfolio solved *51%* of the tasks, that is respectively *23%* and *10%* more than the worst performing configurations and *6%* more than the sequential portfolio.

XCSP In the category XCSP it seems like the tool has hit some kind of limit (the 42 solved tasks are mostly the same) – XCSP is built out of a few basic constraint networks with more and more complex variants added and the complexity of the unsolved tasks

[4]link to exact version used: `https://github.com/ftsrg/theta/tree/2fdf5e06d785bdacb0e278dd3d72a71bd606be46`

[5]link to exact version used: `https://github.com/sosy-lab/sv-benchmarks/tree/99d37c5b4072891803b9e5c154127c912477f705`



Configuration	BitVectors	ControlFlow	ECA	Loops	Sequentialized	XCSP	All	All (%)
expl	12	35	165	83	19	42	**356**	***28%***
pred	17	40	149	244	22	42	**514**	***41%***
sequential-portfolio	24	41	168	257	25	42	**557**	***45%***
complex-portfolio	23	43	185	310	29	42	**632**	***51%***
Number of tasks	**49**	**49**	**190**	**655**	**190**	**117**	**1250**	***100%***

Task category

Figure 5.4: A heatmap showing the number of successfully solved tasks by category for each configuration

grows with such speed that after a certain step, the tool simply hits its performance limits.

ECA, Loops In ECA and Loops the complex portfolio outperforms the others significantly, producing around 8-9% increase in the number of solved tasks even to the second best sequential portfolio. That means that the algorithm selection techniques in the complex portfolio introduced in Section 4.4 are working well (as they are probably used on many of these tasks).

BitVectors Although the category names is BitVectors, there are some tasks that can be handled with integer arithmetics, as shown by the number of tasks solved in *expl* and *pred*. But there is an obvious increase when the bitvector specialized configuration is introduced in the portfolios.

ControlFlow, Sequentialized Both of these categories show a mild increasing trend in the number of successes as we add more and more portfolio and algorithm-selection techniques.

5.2.2 RQ2: Execution Time

How does the complex portfolio perform in terms of CPU time needed compared to the baseline configurations?

5.2.2.1 Average and Total Execution Times

In Figure 5.5 the top column charts show the average execution times per task and the total execution times. The bottom diagrams are similar, except that they are filtered to use the times only from tasks that were successfully solved by the given configuration. Each given time is measured in CPU time and the time limit and thus the maximum value of a single execution on a given task was 900 seconds.

The charts on the left show average CPU times by configurations. The sequential portfolio has the largest average CPU time values, both in the case of including only successful tasks and when including all tasks as well. This is not surprising as although the sequential portfolio is capable of solving many tasks (more than a single configuration), but it might

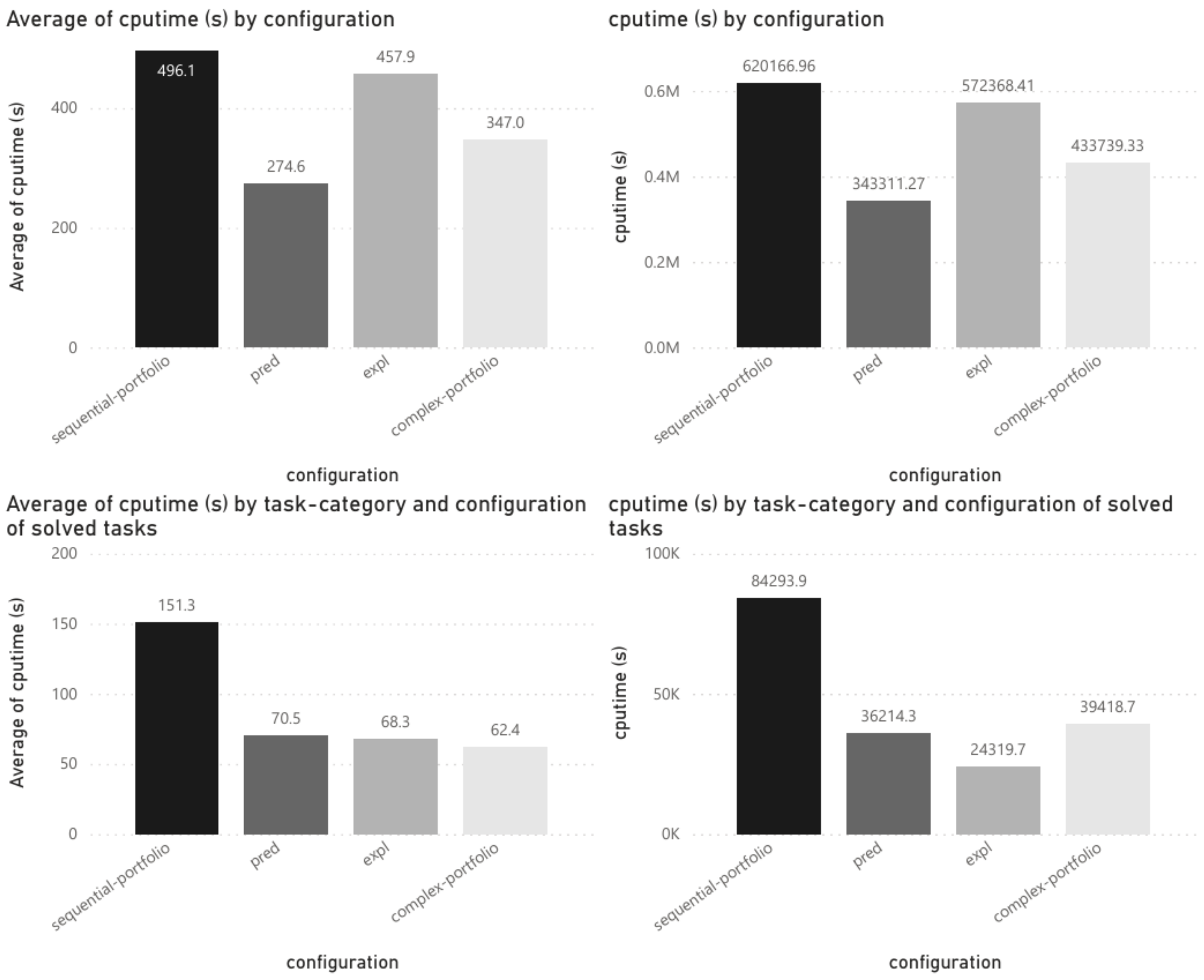


Figure 5.5: Average and total execution times by configuration. The bottom diagrams take only those tasks into account that were successfully solved by the given configuration.

only be capable after the first and maybe even the second configuration of the sequence times out.

The average of the complex portfolio is about the same as the average of the single configurations and is about in the middle inbetween the two if we take all tasks into account. But not significantly better in either case (opposed to the number of successfully solved tasks in Section 5.2.1).

The total execution times are similar to this. It is worth noting that verifying 1250 tasks took about 5 days CPU time in the case of the complex portfolio and about 7.2 days CPU time for the sequential portfolio.

5.2.2.2 Average Execution Times by Category

To further refine the picture about the results, the average CPU times are also shown by category in Figure 5.6.

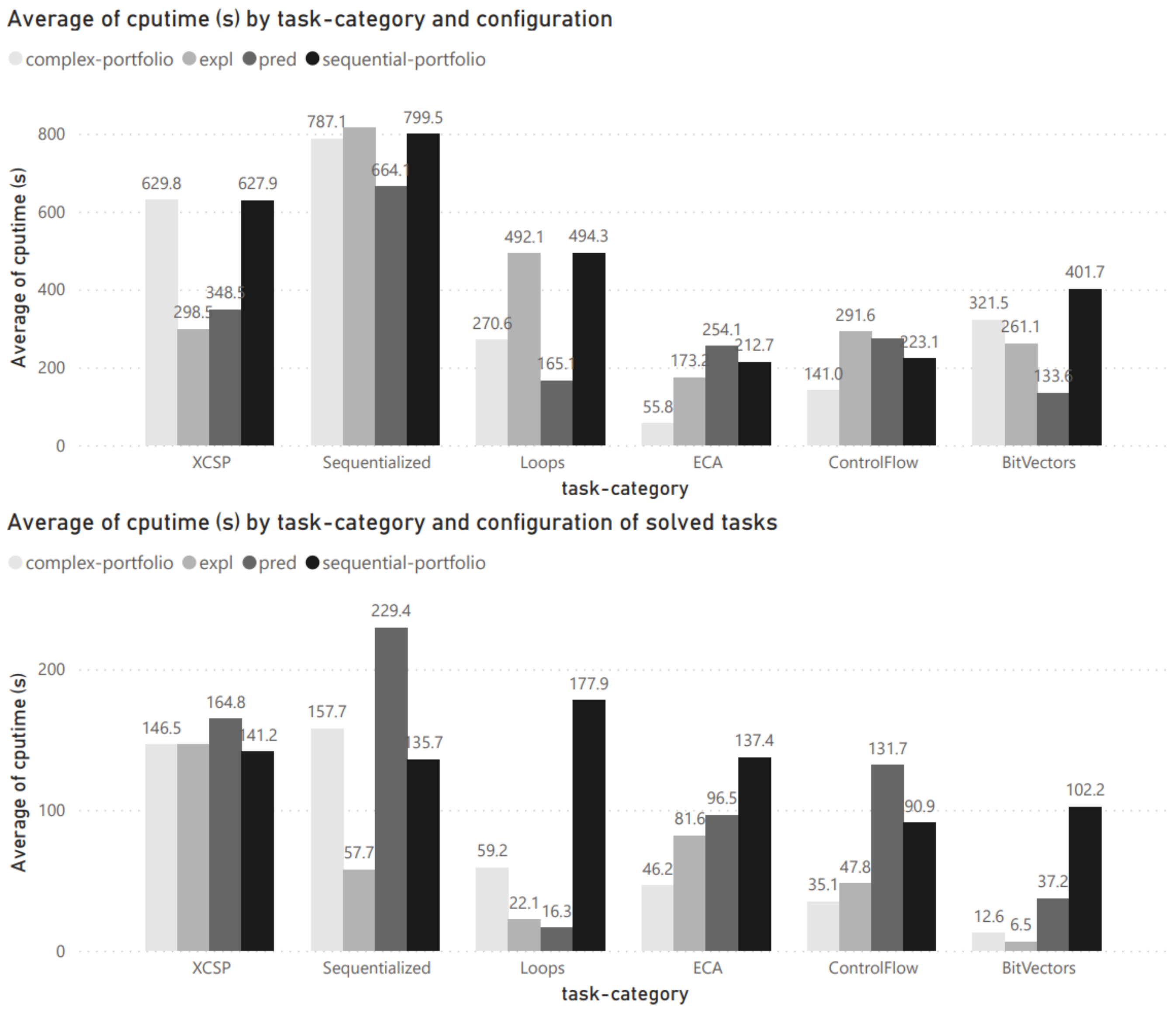


Figure 5.6: Average CPU times by category, taking all tasks into account (above) and counting only successful results (below)

XCSP, BitVectors The complex portfolio performs similarly on XCSP tasks in the successful cases, but gains a significant overhead when the unsuccessful cases are also taken into account. This is because XCSP has tasks that require a configuration capable of handling bitvector arithmetic. In these cases *expl* and *pred* simply throws an error after a while, whereas the portfolios have configurations capable of handling the bitvector arithmetic. Although it is an experimental feature and in its current state ends in a *TIMEOUT* on the XCSP tasks. Of course it would be better to solve these tasks, but not throwing an error in these cases is an improvement as well. The increased time between BitVectors on the bottom and top diagram is also due to the bitvector specific configuration not performing well.

ECA, ControlFlow, Loops In the categories *ECA* and *ControlFlow* the complex portfolio performs really well, and it also outperforms the sequential portfolio in *Loops*. In the case of *ControlFlow* it even outperforms the *expl* configuration and that is probably due to the runtime improvement added in Section 4.3.

Sequentialized In *Sequentialized* it performs similarly to the other configurations, but checking only the successful cases reveals more information – this is probably due to the high number of tasks in this category that time out with every configuration used here. The bottom diagram reveals that in this case the average success time of the *sequential portfolio* is around the same as that of the *complex portfolio.* The explanation is simple: the average time of the *expl* configuration is really low here and it is the first configuration of the *sequential portfolio.*

5.3 Discussion

5.3.1 Conclusion on the Performance of the Complex Portfolio

To summarize, the *complex portfolio* was capable of solving significantly more tasks than the single configurations and was much faster than the *static sequential portfolio.* The latter was made possible due to the dynamic decisions on what configuration to use (Section 4.4) and the time spared on stopping stuck configurations with the runtime improvement (Section 4.3). In other words, it combines the strengths of the different single configurations in a way so that it also reduces the CPU time needed compared to just sequential execution of multiple configurations.

5.3.2 Threats to Validity

In this section the possible biases and threats to validity are discussed. The main points added are on *construct, internal and external validity* [52].

Internal Validity The accuracy of measurements and internal validity are ensured by using Benchexec [14], a state-of-the-art framework developed to execute precisely such benchmarks. Benchexec is able to reliably measure and limit resource usage and uses special containers to isolate the processes measured from others.

Construct Validity Construct validity can be ensured using metrics that can express and measure the examined property, in this case the efficiency and effectiveness of the created dynamic portfolio. The number of successfully solved tasks is a metric used to choose the winners of SV-COMP. That shows that it is a widely accepted metric to measure effectiveness of verification tools. Measuring the time required to verify the tasks is also an important factor in the practical applicability (i.e. it is in most cases not feasible to wait for days to verify a task) and efficiency of the tool.

External Validity External validity, is concerned with how well the results can be generalized. Using the tasks of SV-COMP, which are specially curated to such experiments and are constantly used and evaluated by many different tools and experts of the field gives a good basis for external validity. None the less, threats mainly concern this validity type – the current limits on the capabilites of the frontend (mainly the unsupported C language elements, such as function pointers) exclude many verification tasks, reducing diversity in the set of inputs. Increasing this diversity through adding more and more tasks and task groups could greatly improve the evaluation on the transferability of the portfolio, i.e. its capability to achieve good results on tasks different from the ones we experimented on.

As the difficulty of an input task depends on many factors and is hard to measure, a diverse set of programs to verify is crucial. Despite these problems, the goal of this benchmark is to assess the possibilities of performance enhancement with portfolio-based techniques in the *current version* of the tool Theta and thus the tool's current limitations are an inevitable constraint. Even with this constraint the set of tasks used is fairly large (over a 1000 tasks) and contains diverse categories of the supported language elements. However more diverse and larger benchmarks should be executed in the future to improve external validity of the results.

Chapter 6

Related work

In this chapter I will introduce publications related to this work. The chapter consists of three sections:

- The papers in Section 6.1 are about algorithm selection techniques and tend to concentrate more on what the relevant properties of the input programs are. These usually apply algorithm selection to several tools instead of one.
- Section 6.2 focuses on the portfolios and algorithm selection strategies of different verification tools.
- The last section (Section 6.3) details preliminary works of others and myself on Theta and their connection to this report.

6.1 Algorithm Selection in Software Verification

Variable roles In *Domain Types: Abstract-Domain Selection Based on Variable Usage* [3] the authors introduce the concept of domain types for program variables and develop a pre-analysis step to decide the domain types of all variables in the input program. This is then used to decide the abstract domain between the explicit and BDD domains on a granular, per variable level.

In the paper *On the concept of variable roles and its use in software analysis* [29] a similar concept of variable roles is realized, but creating more roles by adding more practical types. Whilst the first work used its theoretical contributions to add and evaluate a new enhancement to the tool CPAChecker, this one did not yet connect the program properties found to any tool, rather they used machine learning to predict what category the tasks of SV-COMP belong to based on the variables.

Both of these papers are excellent examples on approaching software verification from an algorithm selection standpoint. The general techniques in Chapter 3 pair well with the contributions of these work. In the realization in Chapter 4 I only considered some basic attributes on the variables of a program (mostly if it is an input variable or not), but it is my intention to add more variable properties in the future, similar to these, when they start to make sense in the context of *Theta* (i.e. as the frontend and the analysis begins to handle more variable types).

Machine Learning in Algorithm Selection for Software Verification In *MUX: algorithm selection for software model checkers* [50] the authors use machine learning on available results from different software verifiers to be able to predict, which tool could be successful for a given input program based on structural properties. They collect a list of 131 properties and identify a smaller set of characteristic properties with *feature selection.*

The work *Empirical software metrics for benchmarking of verification tools* [30] adds and studies empirical metrics grouped into three categories: variables, loops and control flow. For each of these groups the authors give deep theoretical foundations in this and earlier work. Based on these properties they also develop a machine learning portfolio solver from the verification tools of SV-COMP, which learns which tool performs best on the training set. Then this portfolio is applied on later years' tasks to demonstrate its predictive power, theoretically winning the *Overall* category.

I shortly discussed in Section 4.2 that although machine learning is not the main focus of this report, the introduced approaches can and hopefully will be used with machine learning techniques in the future. It would be interesting to see how they perform on the more constrained case of using only a single tool instead of creating a portfolio of several tools.

6.2 Verification Tools Using Portfolios and Algorithm Selection

Sequential portfolios with initial algorithm selection *PeSCo* [46][48] is a tool based on *CPAChecker* [10], which uses machine learning techniques to rank several configurations of *CPAChecker* and execute these sequentially in the right order.

VeriAbs [2][27] is a strategy selection-based reachability verifier for C programs. The tool contains four possible pre-defined analysis strategies, which consist of a sequential set of configurations. The preprocessor of the tool chooses a strategy based mainly on the type of loops in the input program and they also check if the given program is *sliceable.* They have found that although classic techniques, such as Bounded Model Checking are capable of solving most of their tasks, the rest of the input programs require a broad range of other techniques to be solved.

CPAChecker [42][10][26] is one of the best known software verification tools built on the foundation of *configurable program analysis.* The framework has many different algorithms implemented, such as k-induction, CEGAR, explicit-value analysis, bounded model checking and so on. They took part in SV-COMP with many different portfolios throughout the years, mostly building around sequential execution of several pre-assembled configurations, but also using algorithm selection on the error property and the input task (e.g. they check if the input program contains recursion, concurrency, loops or complex data)[1].

Each of these tools apply interesting and unique perspectives on how a tool can benefit from portfolios. *PeSCo* is the only one using machine learning. Although each are based on sequential portfolios, *VeriAbs* and *CPAChecker* both wrap the sequential sets into a group or a hierarchy, making it more complex. Opposite to the dynamic portfolio introduced in this work, all of these tools decide on the configurations to use before applying any of them.

[1]This was introduced on SV-COMP'20 and reused on SV-COMP'21: https://gitlab.com/sosy-lab/sv-comp/archives-2021/-/blob/master/presentations/cpa-seq_tacas2021_slides.pdf

Runtime techniques *Ultimate Automizer* [36][37] is a software verifier using an automata-based CEGAR-scheme. They use runtime algorithm selection techniques inside the refinement algorithm, more precisely they are dynamically changing between several methods and SMT solvers for creating the best possible interpolants based on the ones created in the earlier iterations. This is a thoroughly refined runtime technique, but it still fits well into the category of runtime techniques introduced in Section 3.4.3.

6.3 Related Work on Theta

Although this report contains the first dynamic portfolio added to Theta, one of the tool's main attributes is configurability and thus there are earlier works which mainly serve as a basis or foundation to my work.

In *Efficient Strategies for CEGAR-Based Model Checking* [35] many configurational options and algorithmic improvements of Theta were introduced and evaluated with systematic experiments, not just in software, but in model verification as well.

On SV-COMP 2020 the combined tool *Gazer-Theta* [1] used a fairly simple static sequential portfolio of a BMC and two CEGAR configurations – this was a preliminary work to the more complex additions described in this report.

Chapter 7

Conclusion

7.1 Summary of Results

The report made the following theoretical and engineering contributions.

Method Chapter 3 proposed methods for improving the efficiency of configurable model checkers, including techniques on

- sequential portfolio design,
- dynamic configuration selection using the input and earlier results,
- runtime monitoring and intervention in the execution of the verification algorithm,
- assembling a dynamic portfolio.

Realization In Chapter 4 a portfolio is realized in the verification framework Theta. The CEGAR-based dynamic portfolio is assembled for C program verification utilizing most of the techniques of Chapter 3. The process and methods of this tool-specific portfolio assembly are also discussed during this chapter. The portfolio includes a *novel algorithmic improvement* for explicit domain based CEGAR analysis, which is capable of detecting and mitigating when the algorithm makes no progress and gets "stuck" in an infinite loop and the usage of different empirical properties extracted from the input program and the CFA in *algorithm selection*.

Evaluation Lastly in Chapter 5 a *comparative evaluation* was executed to demonstrate how the dynamic portfolio performs, using two *single configurations* and a *simple sequential portfolio* of three configurations as a baseline. The results of the experiment have shown that the dynamic portfolio *outperforms* the single configurations and the static portfolio as well; excelling in *number of solved tasks* compared to the single configurations and in the *execution time* required compared to the static portfolio.

7.2 Future Work

Theta is being developed and improved quickly and shortly there will be new features added that I am planning to include in an improved portfolio. These features include the capability of using several SMT solvers with their own strengths and weaknesses (e.g. improving performance on bitvectors and floating point arithmetic), analysis of concurrent programs, more granular product domains, loop invariants, lazy abstraction.

I would also like to utilize Theta's capability of verification on several formalisms, employing it in two ways:

- adding portfolios that are not specific to software verification,
- enabling Theta to transform the input program or model to more than one formalism and used that as a configuration possibility for algorithm selection.

It would also be important to benchmark on further tasks as support for more and more C language elements is added. This could help both in finding improvement possibilities and in ensuring the transferability and generality of the resulting portfolio, even on less artificial, more realistic input programs and models.

As a long term plan it might also prove useful to try and pair this work with machine learning techniques, such as neural networks when searching for useful program or model properties and their connections to the algorithms' strengths and weaknesses.

Also it would be an interesting direction to apply the general portfolio designing techniques of this work to more verification tools, analyzing the differences between tools from the portfolio viewpoint.

Acknowledgement I would like to thank the support and feedback of Ákos Hajdu.

This research was partially funded by the European Commission and the Hungarian Authorities (NKFIH) through the Arrowhead Tools project (EU grant agreement No. 826452, NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003).

Bibliography

[1] Zsófia Ádám, Gyula Sallai, and Ákos Hajdu. Gazer-Theta: LLVM-based Verifier Portfolio with BMC/CEGAR (Competition Contribution). In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 433–437, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72013-1.

[2] Mohammad Afzal, A. Asia, Avriti Chauhan, Bharti Chimdyalwar, Priyanka Darke, Advaita Datar, Shrawan Kumar, and R. Venkatesh. VeriAbs : Verification by Abstraction and Test Generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1138–1141, 2019. DOI: `10.1109/ASE.2019.00121`.

[3] Sven Apel, Dirk Beyer, Karlheinz Friedberger, Franco Raimondi, and Alexander von Rhein. Domain Types: Abstract-Domain Selection Based on Variable Usage. In *Hardware and Software: Verification and Testing*, pages 262–278. Springer International Publishing, 2013. DOI: `10.1007/978-3-319-03077-7_18`. URL `https://doi.org/10.1007/978-3-319-03077-7_18`.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008. ISBN 978-0-262-02649-9.

[5] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45319-2.

[6] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-36750-5.

[7] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018. DOI: `10.1007/978-3-319-10575-8_11`.

[8] Dirk Beyer. Software verification and verifiable witnesses. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46681-0.

[9] Dirk Beyer. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). pages 401–422. Springer International Publishing, 2021. DOI: `10.1007/978-3-030-72013-1_24`. URL `https://doi.org/10.1007/978-3-030-72013-1_24`.

[10] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

[11] Dirk Beyer and Stefan Löwe. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Fundamental Approaches to Software Engineering*, pages 146–162. Springer Berlin Heidelberg, 2013. DOI: `10.1007/978-3-642-37057-1_11`. URL `https://doi.org/10.1007/978-3-642-37057-1_11`.

[12] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, September 2007. DOI: `10.1007/s10009-007-0044-z`. URL `https://doi.org/10.1007/s10009-007-0044-z`.

[13] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 504–518, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73368-3.

[14] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. 21(1):1–29, November 2017. DOI: `10.1007/s10009-017-0469-y`. URL `https://doi.org/10.1007/s10009-017-0469-y`.

[15] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer Berlin Heidelberg, 1999. DOI: `10.1007/3-540-49059-0_14`. URL `https://doi.org/10.1007/3-540-49059-0_14`.

[16] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-49059-3.

[17] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability.* IOS press, 2009. ISBN 978-1-64368-161-0.

[18] Aaron R Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification.* Springer, 2007. ISBN 978-3-540-74112-1.

[19] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 States and beyond. *Information and Computation*, 98(2):142–170, 1992. ISSN 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(92)90017-A`. URL `https://www.sciencedirect.com/science/article/pii/089054019290017A`.

[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 States and beyond. *Information and Computation*, 98(2):142–170, June 1992. DOI: `10.1016/0890-5401(92)90017-a`. URL `https://doi.org/10.1016/0890-5401(92)90017-a`.

[21] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma

Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing. ISBN 978-3-319-17524-9.

[22] Alonzo Church. A note on the Entscheidungsproblem. *The Journal of Symbolic Logic*, 1(1):40–41, 1936.

[23] 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, September 2003. DOI: 10.1145/876638.876643. URL https://doi.org/10.1145/876638.876643.

[24] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. DOI: 10.1145/186025.186051. URL https://doi.org/10.1145/186025.186051.

[25] Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Predicting Rankings of Software Verification Competitions. *CoRR*, abs/1703.00757, 2017. URL http://arxiv.org/abs/1703.00757.

[26] Matthias Dangl, Stefan Löwe, and Philipp Wendler. CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 423–425, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46681-0.

[27] Priyanka Darke, Sakshi Agrawal, and R. Venkatesh. VeriAbs: A Tool for Scalable Verification by Abstraction (Competition Contribution). In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 458–462, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72013-1.

[28] 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. DOI: 10.1007/978-3-540-78800-3_24.

[29] Yulia Demyanova, Helmut Veith, and Florian Zuleger. On the concept of variable roles and its use in software analysis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, October 2013. DOI: 10.1109/fmcad.2013.6679414. URL https://doi.org/10.1109/fmcad.2013.6679414.

[30] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. Empirical software metrics for benchmarking of verification tools. *Formal Methods in System Design*, 50(2-3):289–316, January 2017. DOI: 10.1007/s10703-016-0264-5. URL https://doi.org/10.1007/s10703-016-0264-5.

[31] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. Craig vs. Newton in software model checking. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 487–497. ACM, 2017. DOI: 10.1145/3106237.3106307.

[32] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792

of *LNCS*, Rome, Italy, March 2013. Springer. URL https://hal.inria.fr/hal-00789533.

[33] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, pages 72–83. Springer Berlin Heidelberg, 1997. DOI: 10.1007/3-540-63166-6_10. URL https://doi.org/10.1007/3-540-63166-6_10.

[34] Orna Grumberg, Doron A Peled, and EM Clarke. *Model checking.* MIT press Cambridge, 1999. ISBN 978-0-262-03883-6.

[35] Ákos Hajdu and Zoltán Micskei. Efficient Strategies for CEGAR-Based Model Checking. *Journal of Automated Reasoning*, 64(6):1051–1091, November 2019. DOI: 10.1007/s10817-019-09535-x. URL https://doi.org/10.1007/s10817-019-09535-x.

[36] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software Model Checking for People Who Love Automata. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 36–52, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8.

[37] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate Automizer and the Search for Perfect Interpolants. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–451, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89963-3.

[38] Falk Howar, Malte Isberner, Maik Merten, Bernhard Steffen, and Dirk Beyer. The RERS Grey-Box Challenge 2012: Analysis of Event-Condition-Action Systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 608–614, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-34026-0.

[39] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated Algorithm Selection: Survey and Perspectives. 27(1):3–45, March 2019. DOI: 10.1162/evco_a_00242. URL https://doi.org/10.1162/evco_a_00242.

[40] F. Kordon, P. Bouvier, H. Garavel, L. M. Hillah, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, , S. Dal Zilio, P. G. Jensen, C. He, D. Le Botlan, S. Li, , J. Srba, . Thierry-Mieg, A. Walner, and K. Wolf. Complete Results for the 2020 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php, June 2021.

[41] Martin Leucker, Grigory Markin, and Martin R 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.

[42] Stefan Löwe, Mikhail Mandrykin, and Philipp Wendler. CPAchecker with Sequential Combination of Explicit-Value Analyses and Predicate Analyses. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 392–394, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54862-8.

[43] 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. DOI: 10.1007/978-3-540-31980-1_1.

[44] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013. ISBN 978-1934356999.

[45] John R. Rice. The Algorithm Selection Problem. volume 15 of *Advances in Computers*, pages 65–118. Elsevier, 1976. DOI: https://doi.org/10.1016/S0065-2458(08)60520-3. URL https://www.sciencedirect.com/science/article/pii/S0065245808605203.

[46] Cedric Richter and Heike Wehrheim. PeSCo: Predicting Sequential Combinations of Verifiers. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–233, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17502-3.

[47] Cedric Richter and Heike Wehrheim. Attend and represent: A novel view on algorithm selection for software verification. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1016–1028, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367684. DOI: 10.1145/3324884.3416633. URL https://doi.org/10.1145/3324884.3416633.

[48] Cedric Richter, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Algorithm selection for software validation based on graph kernels. 27(1-2):153–186, April 2020. DOI: 10.1007/s10515-020-00270-x. URL https://doi.org/10.1007/s10515-020-00270-x.

[49] Tamas Toth, Akos Hajdu, Andras Voros, Zoltan Micskei, and Istvan Majzik. Theta: A framework for abstraction refinement-based model checking. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, October 2017. DOI: 10.23919/fmcad.2017.8102257. URL https://doi.org/10.23919/fmcad.2017.8102257.

[50] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V. Nori. MUX: algorithm selection for software model checkers. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, 2014. DOI: 10.1145/2597073.2597080. URL https://doi.org/10.1145/2597073.2597080.

[51] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58:345–363, 1936.

[52] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-29044-2. URL https://doi.org/10.1007/978-3-642-29044-2.