Bachelor Thesis
in Computer Science

Newton Refinement as Alternative to Craig Interpolation in CPAchecker

Matthias Gerlach

Aufgabensteller:	Prof. Dr. Dirk Beyer
Betreuer:	Dr. Marie-Christine Jakobs, Dr. Philipp Wendler
Abgabedatum:	19.12.2018

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.
This paper was not previously presented to another examination board and has not been published.

München, 19.12.2018

.....
Matthias Gerlach

Abstract

Driven by the growing relevance of software in all kinds of areas, Software Model Checking is an important and active field of research.

One common technique of software model checking is counterexample-guided abstraction refinement(CEGAR). CEGAR often uses Craig Interpolation to extract state assertions from error traces. Yet Craig Interpolation is not supported by all solvers and not applicable for all problems.

Therefore, we introduce Newton Refinement as an alternative approach to extract such state assertions. As a part of this work we implement Newton Refinement in CPAcheckers predicate analysis and evaluate its impact based on a broad set of benchmarks. The results show that Newton Refinement presents an alternative to interpolation. Especially, it allows to verify programs where interpolation is no option.

Contents

List of Figures

List of Tables

1. Introduction

Today computers - and with them programs - are everywhere. We use programs not only for communication, work and entertainment but also in devices relevant to our security such as cars and airplanes.

The more we depend on programs in our lives, the more important it is that these programs work as expected. A failing program no longer just means a crashed computer, but could eventually lead to millions of euros of damage or even worse serious injuries or death.

Therefore, today more than ever it is necessary to find ways to prove the correctness of programs. Thus *Software Model Checking*, the algorithmic analysis to prove the properties of a program, is an important and active field of research.[4, 3, 12, 11]

One common technique of software model checkers is *Predicate Abstraction*[11]. The idea of predicate abstraction is to reduce the size of the model by including only those assertions in the model which are necessary to prove the programs correctness. Predicate abstraction is often based on counterexample-guided refinement (CEGAR)[8]. CEGAR uses infeasible error traces found by a reachability analysis to improve the model. To do so CEGAR needs a method to extract state assertions from the error trace. These state assertions are used as predicates in the model. One of the methods to create such state assertions from an error trace is Newton Refinement[1].

The goal of this thesis is, to implement Newton Refinement into CPAchecker[5] and evaluate its impact. Newton Refinement, named after the NEWTON tool is a method to refine a given error trace to a sequence of state assertions. Many recent tools use Craig Interpolation[9] for the refinement of error paths[2]. Historically Newton Refinement is the predecessor of Craig Interpolation and was replaced under the assumption that Craig Interpolation creates better predicates.

In a 2017 paper called "Craig vs. Newton in Software Model Checking"[10], the Newton algorithm was revisited and implemented in the Ultimate framework[1]. The paper suggests, that Newton Refinement achieves results com-

[1] https://ultimate.informatik.uni-freiburg.de

parable to those of Craig Interpolation and recommends further research of the method. Based on this paper, we implemented Newton Refinement in the CPAchecker framework and performed a wide range of benchmarks. Another reason why Newton Refinement is interesting is the relatively small number of mature SMT solvers that support Interpolation. With Z3 dropping support of interpolation this is becoming even more an issue [2].

In Sect. 2, we explain the background of CEGAR based predicate analysis. The next section(Sect. 3) introduces the methods and optimizations used for Newton Refinement. The implementation into CPAchecker is the topic of Sect. 4. In Sect. 5 we will perform a number of benchmarks of CPAcheckers predicate analysis using Newton Refinement. We compare several configurations of Newton Refinement in order to find the best configuration(Sect. 5.1). The influence of the chosen SMT solver on the results of Newton Refinement will be analyzed in Sect. 5.2. Finally, in Sect. 5.3 we compare the best Newton Refinement configuration with the interpolation-based predicate analysis of CPAchecker.

[2]https://github.com/Z3Prover/z3/pull/1646

2. Counterexample-Guided Abstraction Refinement based Predicate Analysis

One problem of model checking algorithms that create the entire state space is that the number of states becomes huge or even infinite. In order to overcome this state explosion, Predicate Analysis[11] overapproximates the state space with abstract states. Each abstract state covers a number of concrete states so the overall number of states can be reduced. An abstract state is a tuple of a location within the program and a set of predicates. The predicates are logical formulas that constrict the possible values of the program variables.

One common technique to create a predicate abstraction is *Counterexample-Guided Abstraction Refinement* (CEGAR)[8]. CEGAR is based on the idea that if we are able to find information excluding all error paths from the set of feasible paths the program is safe.

The goal of CEGAR is to create an abstraction that overapproximates the reachable program states. At the same time, this abstraction has to be concrete enough to prove the unreachability of all error locations. Based on the error traces found by a data-flow analysis, CEGAR creates predicates over the set of program variables. The predicates overapproximate the set of feasible states at a program location.

The basic approach of CEGAR is to iteratively refine a model based on error paths found by the data-flow analysis. Starting with a coarse overapproximation, the abstract model of the program is steadily defined more precisely. The CEGAR algorithm ends when all error states are excluded from the reachable program states or a feasible counterexample is found.

CEGAR uses a reachability analysis of the program states creating a representation of the reachable state space of the program. Mature model checkers such as CPAchecker create an *Abstract Reachability Graph*, incorporating methods such as *Adjustable Block Encoding*[6] and much more. For sake of simplicity, we will only consider a simple case where each state is only represented by its

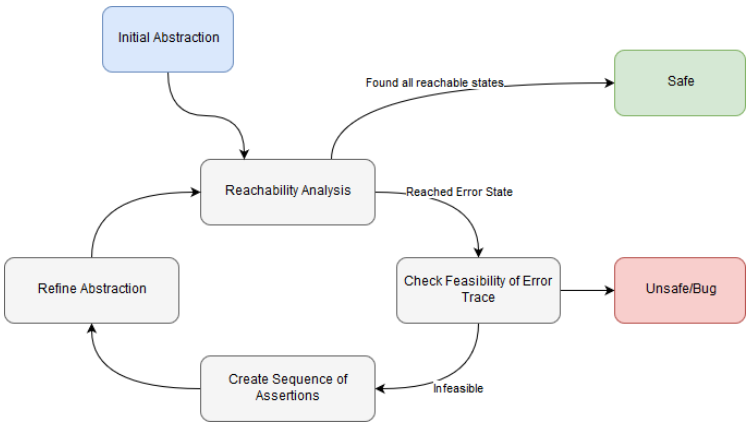

Figure 2.1: Steps of Counterexample-Guided Abstraction Refinement

location and the predicates constricting the program variables. These predicates are the result of previous iterations of CEGAR. The set of predicates of an abstract name is named precision. To present such a state space and the paths within we will use a reachability tree.

The steps of CEGAR are presented in Fig. 2.1. In the beginning the analysis is started using an empty precision, creating a reachability tree without any assertions about the program variables. Once the analysis reached an error state, the trace to this state is reconstructed from the reachability tree. If the error trace is feasible, CEGAR will return the trace. This means, that the error is reachable and the trace shows the existence of a bug. If, however, the error trace is infeasible, the abstraction is not precise enough to exclude this trace. The model has to be refined in such a way that this error trace is no longer reachable.

To refine the abstraction of the program, we first need to find a sequence of assertions which excludes the trace from the reachability tree. These logical formulas are the facts required to prove the infeasibility of the path. A commonly used way to find such a sequence is *Craig Interpolation*[9], but in this thesis we will concentrate on an alternative method called *Newton Refinement*[1], which will be explained in detail in Sect. 3.

The assertions found by either Craig Interpolation or Newton Refinement are merged with the previous precision to create a new precision, which is precise enough to cover the infeasible error path from the reachability tree.

The reachability analysis is restarted with the new precision and the previous steps will be performed repeatedly until either a feasible error path is found or the abstract model is proven safe. The model is safe when no more states can be found by the reachability analysis. As all previously found error paths are infeasible and now excluded by the precision, no error state is reachable.

3. Newton Refinement

In this section we will introduce Newton Refinement. Newton Refinement is an approach to abstract an error trace in order to find a sequence of assertions proving its infeasibility. The name Newton Refinement originated from the tool NEWTON[1], where it was first implemented. In the following we will define the Newton approach similar as described by Dietsch et al.[10].

We will use the example program displayed in Fig. 3.1 to explain the different notations and methods. The program is safe if the `ERROR` label is unreachable. The only relevant variable is x, which is first assigned to $x = 0$ in line 2. The variable y is not necessary for the program logic but serves as an example for the live-variable abstraction explained later on. As one can see the while loop increments x by one. After the first iteration $x = 1$ violates the loop condition $x < 1$ and the loop will not be executed a second time. The error label can only be reached if $x \neq 1$, which is trivially false for $x = 1$. Thus, the program is save.

Next to the program you can find the corresponding reachability tree (Fig. 3.2). The reachability tree consists of a number of states that can be found by a reachability analysis. Each state is described by a tuple (l, p) where l denotes the line in the source code and p is a predicate. The predicates presented in the tree are an example how the final set of predicates could look like. The edges between the states are labeled with the corresponding assume or assignment statements. These statements are simplifications of the exemplaric C-code. The tree also depicts two error traces that could be subject of a Newton Refinement. One error path is colored in red and the other is colored in blue. Both end in the error location(line 9: `goto ERROR;`).

First, we will define some notations and terminology used to describe the Newton Refinement in a more formal fashion.

A program run can be described by its sequence of statements, we will call such a sequence trace $\tau = st_1, \ldots, st_n$. If a trace ends in an error state we call it error trace. In order to simplify the traces we will use only two types of statements.

```
1  int main(){
2    int x = 0;
3    int y = 1;
4
5    while(x < 1){
6      x = x + 1;
7    }
8    if (x != 1){
9      goto ERROR;
10   }
11   return 0;
12 ERROR:
13   return -1;
14 }
```

Figure 3.1: Source-code of the example program

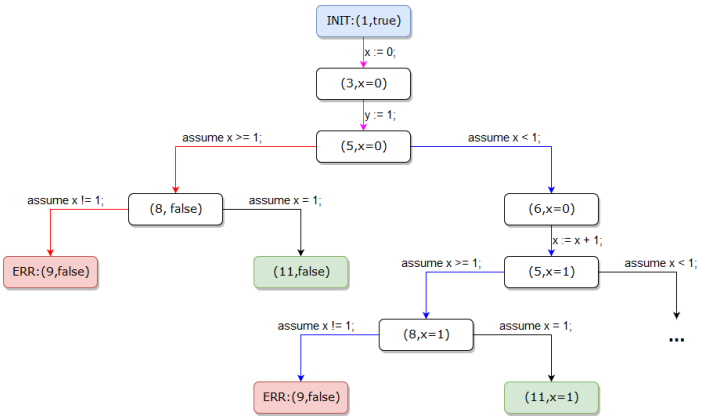

Figure 3.2: Reachability Tree of the example program

1. `assume` φ: An assumption over the set of program variables, where φ is a boolean expression over the set of program variables. An assumption will not affect the values assigned to the program variables.

2. $x := e$: An assignment of the program variable x to an expression e over the set of program variables. The variable x is assigned to a new value.

A possible error trace for the example program can be found in the first column of 3.1. It represents the red trace in the reachability tree(cf. Fig. 3.2). Both $x := 0$ and $y := 1$ are examples for assignments. The trace forms a case where x does not fulfill the loop condition $x < 1$. Therefore it can be abstracted by `assume` $\neg(x < 1)$. The second assume `assume` $x \neq 1$ states that the if-statement is fulfilled and the error is reached.

Each statement can be described by a path formula $F_i = F(st_i)$, a first-order formula over the program variables describing the logical implication of a single statement. As a program variable may be assigned several times in one trace, it will be substituted by its indexed form using the formula $\text{rename}_i(\varphi)$. The index i specifies the location of the statement in the trace.

$$F(st_i) = \begin{cases} \text{rename}_i(\varphi) & \text{if } st_i \text{ is } \texttt{assume } \psi \\ x_i = \text{rename}_i(e) & \text{if } st_i \text{ is } x := e \end{cases} \tag{3.1}$$

The function $\text{rename}_i(\varphi)$ replaces each program variable x in the formula φ by its indexed form x_k, where k is the position of the last assignment. For this

Table 3.1: Sequence of Assertions Strongest Postcondition

i	st_i	F_i	State assertion φ_i
			$\varphi_0 := \texttt{true}$
1	$x := 0;$	$x_1 = 0$	
			$\varphi_1 := x_1 = 0$
2	$y := 1;$	$y_2 = 1$	
			$\varphi_2 := x_1 = 0 \wedge y_2 = 1$
3	`assume` $\neg(x < 1);$	$x_1 \geq 1$	
			$\varphi_3 := x_1 = 0 \wedge y_2 = 1 \wedge \neg(x_1 < 1)$
			$\Rightarrow \varphi_3 := \texttt{false}$
4	`assume` $x \neq 1;$	$x_1 \neq 1$	
			$\varphi_4 := \texttt{false}$

purpose we define $k = \text{index}(x, i)$, that determines the last location where the variable x has been modified.

$$\text{index}(x, i) = \begin{cases} 0 & \text{if } i = 0 \\ i & \text{if } st_i \text{ is } x := e \\ \text{index}(x, i-1) & \text{else} \end{cases} \tag{3.2}$$

The conjunction of all path formulas of a trace is called trace formula F:

$$F = F(st_1) \wedge F(st_2) \wedge \ldots \wedge F(st_n) \tag{3.3}$$

Based on the satisfiability of the trace formula we can decide whether a trace is feasible or not. If the trace formula is unsatisfiable, the trace is infeasible and a refinement is possible. If the formula F is satisfiable, the trace is a counterexample proving an error within the program.

Returning to our example, the path formulas are depicted in the second column of Tab. 3.1. To show the application of renaming, we will explain it using statement st_4(`assume` $x \neq 1$). The assume statement contains the program variable x which has to be replaced by its indexed form. x has last been modified in st_1 , so we substitute $(x \neq 1)[x \rightarrow x_1]$ and obtain the path formula $F_4 = x_1 \neq 1$.

The output of Newton Refinement is a sequence of assertions $\varphi_0, \ldots, \varphi_n$ for the trace $\tau = st_1, \ldots, st_n$. The sequence proves the infeasibility of the trace. Each sequence of assertions has to fulfill following conditions:

$$\varphi_0 = \texttt{true} \tag{3.4}$$

$$\varphi_{i+1} = \mathbf{SP}(\varphi_i, st_{i+1}) \qquad \text{for } i = 0, \ldots, n-1 \tag{3.5}$$

$$\varphi_n = \texttt{false} \tag{3.6}$$

The first assertion, corresponding to the location prior to the first statement of the trace, always has to be true as no assertion can be made at this point.

The last assertion is the postcondition of the whole trace and is always false, otherwise the error trace would be feasible and the refinement would never be started. All assertions in between the first and last assertion are derived by iteratively performing the strongest postcondition operation on each statement. The precondition we use is the postcondition of the previous statement. The necessary operations to compute the strongest postcondition are described in the following paragraph.

We want to create an inductive sequence of assertions, where each assertion is stronger than the one before. Therefore we iterate over the program trace and compute the strongest postcondition for each statement using the formula ϕ as precondition. ϕ represents the strongest postcondition of the predecessor. The first assertion is always `true` (cf. Eq. 3.4), so the precondition ϕ of the first statement is also `true` .

The strongest postcondition operation can be defined for our two types of statements as:

$$\mathbf{SP}(\phi, x := e) = \exists x_j . F(x := e) \wedge \phi \tag{3.7}$$

$$\mathbf{SP}(\phi, \texttt{assume}\ \psi) = \phi \wedge F(\texttt{assume}\ \psi) \tag{3.8}$$

The strongest postcondition operation is basically a conjunction of the precondition ϕ and the path formula $F(st_i)$ of the statement st_i. In the case of an assignment we are additionally able to quantify the previous assignments of x. This may be done as all following references will only reference the new indexed variable. The quantified variable x_j references the previous assignment of x. To remove the quantified formula from the assertion we can use quantifier elimination.

In our example trace the results are presented in the third column of Tab. 3.1. This trace does not contain any reassignment, so no quantification has to be applied. The assertion φ_3 is already unsatisfiable and can be substituted with `false` as it contains the contradiction $x_1 = 0 \wedge x_1 \geq 1$. The resulting sequence of assertions fulfills all conditions defined before.

3.1 Path-Formula Abstraction using an Infeasible Core

Let formula Φ be a unsatisfiable conjunction of a set of formulas $\{F_1, \ldots, F_n\}$. UC is an *Unsatisfiable Core* of Φ if $UC \subseteq \{F_1, \ldots, F_n\}$ holds and the conjunction of the subset still is unsatisfiable. An unsatisfiable core has to be neither minimal nor unique, and the set of all formulas is an unsatisfiable core itself.

Table 3.2: Sequence of assertions using infeasible-core abstraction

i	Program Trace st_i	Path Formula F_in	Strongest Postcondition with IC
			$\varphi_0 := \texttt{true}$
1	$x := 0;$	$x_1 = 0$	
			$\varphi_1 := x_1 = 0$
2	$y := 1;$	$y_2 = 1$	
			$\varphi_2 := x_1 = 0$
3	`assume` $x < 1;$	$x_1 < 1$	
			$\varphi_3 := x_1 = 0$
4	$x = x + 1;$	$x_4 = x_1 + 1$	
			$\varphi_4 := \exists x_1.x_1 = 0 \wedge x_4 = x_1 + 1$
			$\Rightarrow \varphi_4 := x_4 = 1$
5	`assume` $\neg(x < 1);$	$x_4 \geq 1$	
			$\varphi_5 := x_4 = 1$
6	`assume` $x \neq 1;$	$x_4 \neq 1$	
			$\varphi_6 : x_4 = 1 \wedge x \neq 1$
			$\Rightarrow \varphi_6 := \texttt{false}$

We will use the concept of an unsatisfiable core in the context of state assertions to obtain the infeasible core. As an unsatisfiable trace formula F(cf. Eq. 3.3) indicates an infeasible path, we name an unsatisfiable core of the trace formula *Infeasible Core* ($IC(F)$). So when we compute the unsatisfiable core of the trace formula, we get a subset of path formulas which are required to make the trace infeasible.

Now we will define the strongest postcondition in a way that uses the infeasible core for abstraction. Only path formulas that are part of the infeasible core will be added to the post condition. All other path formulas are abstracted. An abstracted assignment still changes the value of the program variable x. Therefore we can quantify the old indexed variable x_j.

$$\mathbf{SP}_{IC}(\varphi, x := e) = \begin{cases} \exists x_j.F(x := e) \wedge \varphi & \text{if } F(x := e) \in IC(F) \\ \exists x_j.\varphi & \text{else} \end{cases} \tag{3.9}$$

$$\mathbf{SP}_{IC}(\varphi, \texttt{assume } \psi) = \begin{cases} \varphi \wedge \psi & \text{if } F(\texttt{assume } \psi) \in IC(F) \\ \varphi & \text{else} \end{cases} \tag{3.10}$$

In order to show the changed behavior of the strongest postcondition when using the infeasible-core abstraction, we present a different trace of the example before. The refinement is performed on the error trace as displayed in the first column of Tab. 3.2. The trace is highlighted in the reachability tree (cf. Fig. 3.2) with a blue color. The path formulas of the trace are presented in the second column and we obtain the set of all path formulas F.

$$F = \{x_1 = 0, y_2 = 1, x_1 < 1, x_4 = x_1 + 1, \neg(x_4 < 1), x_4 \neq 1\}$$

Now we suppose that we obtain the following infeasible core $IC(F)$

$$IC(F) = \{x_1 = 0, x_4 = x_1 + 1, x_4 \neq 1\}$$

Based on the infeasible core, we are now able to compute the strongest post-conditions using the definitions 3.9 and 3.10. The calculated strongest post-conditions are presented in the third column of Tab. 3.2. One can see that no longer every path formula, but only those formulas contained within the infeasible core add to the sequence of assertions. This results in shorter and less specific assertions, whereas the last assertion is still unsatisfiable.

3.2 Quantification of Live Variables

We call a variable *Live Variable* at position i of the trace iff it is read in any following statement st_j(where $j > i$). The concept of live variables is used to eliminate variables, which are not future live from the strongest postconditions. They do not add any information necessary for making the trace infeasible and can be safely quantified.

To discover variables that are not future live in a specific assertion, we use the following procedure. Starting with the sequence of assertions, all variables present in the sequence of assertions are mapped to the last path formula they are part of.

$$last\_read(x) = last\_read(x, n) \tag{3.11}$$

Where n is the index of the path formula and x is some program variable. We define a helper function $last_{pos}(x, i)$ to create a recursive function returning the index of the last statement where x was used.

$$last\_read(x, i) = \begin{cases} 0 & \text{if } i = 0 \\ i & \text{if } x \text{ occurs in } F_i \\ last\_read(x, i-1) & \text{else} \end{cases} \tag{3.12}$$

When we apply $last\_read(x)$ on a program variable we obtain the last position it was read. We can now divide the program variables into two sets for each location i. The set of variables which are future live (FL_i) and those which are not ($\overline{FL_i}$).

For the set of variables not being future live ($\overline{FL}$) holds:

$$\forall x \in \overline{FL_i}.last\_read(x) \leq i \tag{3.13}$$

Table 3.3: Sequence of assertions using Live Variables

i	st_i	F_i	State assertion φ_i	FL_i	$\overline{FL_i}$
			$\varphi_0 := \texttt{true}$		
1	$x := 0;$	$x_1 = 0$		$\{x_1, y_2\}$	$\{\}$
			$\varphi_1 := x_1 = 0$		
2	$y := 1;$	$y_2 = 1$		$\{x_1\}$	$\{y_2\}$
			$\varphi_2 := \exists y_2.x_1 = 0 \wedge y_2 = 1$		
			$\Rightarrow \varphi_2 := x_1 = 0$		
3	$\texttt{assume } \neg(x < 1);$	$x_1 \geq 1$		$\{x_1\}$	$\{y_2\}$
			$\varphi_3 := x_1 = 0 \wedge \neg(x_1 < 1)$		
			$\Rightarrow \varphi_3 := \texttt{false}$		
4	$\texttt{assume } x_1 \neq 1;$	$x_1 \neq 1$		$\{\}$	$\{x_1, y_2\}$
			$\varphi_4 := \texttt{false}$		

And in turn for the set of future live variables (FL) holds:

$$\forall x \in FL_i.last\_read(x) > i \tag{3.14}$$

Variables which are not future live will be existentially quantified and therefore removed from the state assertion. So we define the future live operation as:

$$\mathbf{LV}(\varphi_i) = \exists \nu_0, \ldots, \nu_n.\varphi_i \qquad \text{with } \{\nu_0, \ldots, \nu_n\} = \overline{FL_i} \tag{3.15}$$

Returning to our example, we look at the red trace in Fig. 3.2. We present the trace in more detail in Tab. 3.3. The statements are presented in the first column of the figure. The corresponding path formulas are displayed in the second column. The third column shows the state assertions for each state. Finally the fourth and fifth column present the sets of future-live variables FL_i and its complement $\overline{FL_i}$.

As the variable y is assigned in the second statement and never used later on, it is a perfect example for a variable which is not future live. The last position of the indexed variable y_2 is computed to be $last\_read(y_2) = 2$ and therefore y_2 is not future live in any position $i \geq 2$. So we are able to existentially quantify y_2 in φ_2. The variable x_1 is future live in all positions except for the last. It could be quantified in φ_4, but as a quantification on `false` has no effect it is omitted.

4. Implementations in CPAchecker

The algorithm for Newton Refinement was implemented into *CPAchecker*[5], a state of the art software verification tool developed by *SosyLab*. CPAchecker combines various program analysis and model checking algorithms in one framework.

In detail, the Newton Refinement was implemented to extend the functionality of the predicate analysis included in CPAchecker. It adds an alternative to the default error trace refinement based on Craig Interpolation.

Newton Refinement depends on quantifier elimination for creating high-quality predicates. As Quantifier Elimination is not supported by all SMT-solvers, an additional module performing a best effort quantifier elimination was developed. In the following, it will be called `PseudoQuantifierElimination`.

4.1 Implementation of Newton Refinement in CPAchecker

The core of the Newton refinement was implemented in the `NewtonRefinementManager` class. It is build upon the already implemented predicate analysis of CPAchecker and reuses existing components. CPAchecker represents programs by a *control-flow automaton*(CFA), a directed graph where the edges(`CFAEdge`) represent statements of the program. Each `CFAEdge` has a corresponding `Pathformula`.

CPAcheckers reachability analysis creates an *Abstract Reachability Graph* (ARG), a directed graph representing all reachable states. CPAchecker reconstructs the error trace from the ARG once an error state is found. Afterwards it uses Newton Refinement to create a sequence of assertions from this error trace.

One of the core parts of Newton Refinement is the creation of the strongest postcondition at the error trace locations. In this thesis we developed two

independent methods to compute the strongest postconditions of the trace. The first method is implemented to be close to the approach described by Dietsch et. al.[10] and we will call it edge level abstraction. It performs the abstraction at each `CFAEdge` of the error trace. The second method called block-level abstraction uses the `Blockformulas` provided by the *Adjustable-Block Encoding* [6]. Adjustable-block encoding is used to encode a configurable number of statements in one logical formula. The resulting formulas are called `Blockformulas` as they encode a block of edges instead of a single edge. We name the states where a Block starts or ends *abstraction state*.

In order to abstract the postconditions, we implemented the infeasible-core and the live-variable abstraction. Both optimizations were previously introduced in Sect. 3.1 and Sect. 3.2.

By default, Newton Refinement is deactivated within the CPAchecker predicate analysis. To enable Newton Refinement you have to add the following option to your configuration file:

```
cpa.predicate.refinement.useNewtonRefinement = true
```

In the following, we will describe the implementation of the different methods in more detail and will also describe which configuration options have to be set to use a specific method.

4.1.1 Implementation of the Edge-Level Abstraction for Newton Refinement

As mentioned before, the edge-level abstraction is implemented in a way to be close to the methods described in Newton vs. Craig [10]. It is based on the path formulas of each edge in the control-flow automaton instead of the block formulas derived by the optimized reachability analysis.

Each statement in the error trace results in a `CFAEdge` in the CFA. A `CFAEdge` holds information about the type(`CFAEdgeType`) of the statement it represents. Based on the type, we map each edge type to one of our statement types described in chapter 3.

- An `CFAEdge` will be treated as an assume statement if its `CFAEdgeType` is `AssumeEdge`.
- If the `CFAEdgeType` is one of {`StatementEdge`, `DeclarationEdge`, `FunctionCallEdge`, `ReturnStatementEdge`, `FunctionReturnEdge`} it will be considered an assignment.

In order to have a simple way to access all required information to a position within the trace, we introduce the helper class `PathLocation`. A `PathLocation` holds information about the position of a state within the trace, the

incoming `CFAEdge`, its `Pathformula`, and whether it is an abstraction state. To represent the whole trace we create a list of `PathLocations` from the initial state to the error state.

To create the sequence of assertions we loop over the `PathLocations` and create the postcondition for each `CFAEdge` by applying the rules introduced in Sect. 3. For an assignment, we will first create the conjunction of the postcondition of the previous edge and the path formula of the edge itself. Within the resulting formula, we will now existentially quantify the indexed variables which have been assigned to a new value. Afterwards, we try to eliminate the quantified variables from the formula by using the `PseudoQuantifierEliminationManager` introduced in chapter 4.2.

We start with the initial assertion `true` for the initial state. The predicate analysis only requires assertions for abstraction states, but we create the postcondition of each `CFAEdge`. Therefore we have to detect if a `PathLocation` has a corresponding abstraction state and only then do we add its strongest postcondition to the sequence of assertions. After we iterated over all trace locations, we obtain the final postcondition which is the postcondition of the whole error trace. As the error trace is infeasible, otherwise the refinement would not be performed, this last assertion needs to be `false` (cf. Eq. 3.6). If it is not, we return an exception stating that our algorithm was not able to obtain a strong enough assertion. This may happen as not every quantifier elimination is successful. If the final postcondition is strong enough, we return the sequence of assertions.

Edge-level abstraction and block-level abstraction are mutually exclusive settings. Edge-level abstraction is the default configuration but can also be specified in the configuration file with following option.

```
cpa.predicate.refinement.newtonrefinement.abstractionLevel = EDGE
```

4.1.2 Implementation of the Block-Level Abstraction for Newton Refinement

The block-level abstraction is a more high level approach to the methods described in Newton vs. Craig [10]. Instead of calculating the postconditions for each edge of the automaton we use the block formulas calculated between abstraction states in the abstract reachability graph. These block formulas are an optimization, as they cover a larger part of the reachability tree.

`BlockFormulas` between the abstraction states are treated similar to the path formulas used in Sect. 3. Instead of quantifying a variable at each point it is assigned a new value, we will abstract all variables with an old index at

each abstraction state. In this case, an old index means that the index is not equal to the last location where the variable has been assigned. CPAchecker already holds these information about the variable indices in the path formulas `SSAindex`, so that variables with an old index can be easily detected and existantially quantified. In order to quantify and eliminate the variables we again use the `PseudoQuantifierEliminationManager`.

Block-level abstraction can be activated with following configuration setting.

```
cpa.predicate.refinement.newtonrefinement.abstractionLevel = BLOCK
```

4.1.3 Implementation of the Infeasible Core Abstraction

One way to abstract the strongest postcondition to a helpful predicate is an abstraction based on the unsatisfiable core of the trace formula. The trace formula is the conjunction of all path formulas of the trace. For block-level abstraction, the path formulas of the blocks between abstraction states are already computed and provided as a parameter. For edge-level abstraction, the path formulas can be obtained from the list of `PathLocations`.

The unsatisfiable core of the trace formula is calculated by the SMT solver and the result is the infeasible core of path formulas. The infeasible core is a subset of the path formulas.

We use this set of formulas to determine whether a statement, and respectively its path formula, can be abstracted while performing the strongest postcondition operation. We test for each path formula(respectively block formula for block-level abstraction) if it is part of the infeasible core. If it is not, the path formula will not be conjuncted to the postcondition of the previous block. It does not add any information that is necessary for the infeasibility of the trace formula. Nonetheless, if an assignment is performed within the block or edge, the old indexed variables will be quantified.

The infeasible core abstraction is active by default but can be deactivated with a configuration option:

```
cpa.predicate.refinement.newtonrefinement.infeasibleCore = false
```

4.1.4 Implementation of the Live Variable Abstraction

The live-variable abstraction is implemented as a post processing on the predicates to keep it independent from the code for strongest postcondition and infeasible core.

First, we create a set containing all variables which occur within the predicates. Then we loop over the trace locations to create a map, where each variable is mapped to the path location where it is last read. Based on this information we can now filter the variables in each predicates, whether they are relevant for following assertions or not. This can be easily done by comparing the last location where the variable is read with the location of the predicate.

If the variable is not read in following path formulas it will be added to the set of variables that can be safely quantified. Once all variables within a predicate are filtered, all variables which are not future live will be existentially quantified. Then we try to eliminate them using the `PseudoQuantifierEliminationManager` described in Sect. 4.2.

Using live variables is the default behavior. To deactivate live-variable abstraction following configuration option may be added:

```
cpa.predicate.refinement.newtonrefinement.liveVariables = false
```

4.2 Implementation of a Solver Independent Quantifier Elimination

Quantifier elimination is an important operation for the quality of the predicates obtained by Newton Refinement, but several solvers including the solver MathSAT5[1], best supported by CPAchecker, do not support it.

As there exist some simple methods to perform quantifier elimination on conjunctions, a module to utilize these methods is a possible improvement for Newton Refinement. In particular these methods are described in the paper Newton vs. Craig[10].

The first way of simplification is *Destructive Equality Resolution*, abbreviated as DER. If there is a clause which asserts the equality of the variable and an arbitrary expression, this variable can be substituted by this expression in all other parts of the conjunction. Or more formally $\exists x.x = t \wedge \varphi$ is equivalent to $\varphi[x \mapsto t]$ if t does not contain x.

To achieve this, the quantified formula is sorted in conjuncts containing quantified variables and those not containing any quantified variables. The next step is to find conjuncts defining an equality where one of the arguments is a quantified variable. Once such an equality is found, all other occurrences of the quantified variable are replaced by the term it is equal to.

[1] http://mathsat.fbk.eu

DER is activated by default, yet it can be disabled with the following configuration option:

```
cpa.predicate.pseudoExistQE.useDER = false
```

The second simplification strategy used is the *Unconnected Parameter Drop*, or UPD. If we find a satisfiable conjunct only containing quantified variables and theory axioms(e.g. numbers, boolean values), we can drop this conjunct as it does not influence any other part of the formula.

Similar to DER, UPD is also enabled by default but can be deactivated by setting:

```
cpa.predicate.pseudoExistQE.useUPD = false
```

Additionally the module implements a way to use the quantifier elimination offered by the solver. If such a functionality is available, it can be used to quantify variables which cannot be eliminated by the simplification techniques explained above.

By default, the `PseudoQuantifierEliminationManager` will use best effort quantifier elimination($LIGHT$). But it can also be configured to not use the solver($NONE$) or to use the full quantifier elimination ($FULL$), which may result in very long run times. Not all solvers support quantifier elimination. For the solvers not supporting it, this option is always set to $NONE$:

```
cpa.predicate.pseudoExistQE.solverQeTactic = [NONE,LIGHT,FULL]
```

Quantifier elimination is not always successful as the implemented methods are only a best effort way of removing quantified variables from the formula. Furthermore, not all solvers support quantifier elimination and even if they do it may fail. To avoid returning a still quantified formula it is possible to overapproximate the formula by dropping all conjuncts containing any of the quantified variables. The resulting formula is weaker than the result of a quantification, but it may be helpful to have such a weaker formula.

Overapproximation is turned off by default. To activate the functionality one can add the following option to the configuration file:

```
cpa.predicate.pseudoExistQE.overapprox = true
```

4.3 Implementation of a Fallback to Interpolation for Newton Refinement

As described before, the refinement will not always find a strong enough formula to exclude the analyzed error trace. To be still able to continue the analysis of the program it is possible to use the existing interpolation-based

refinement as a fallback. Due to the modularized structure of the software, this can be done quite easily. A fallback will occur either if the last predicate is not strong enough to exclude the error trace and an exception is raised or if an error trace is found a second time.

Fallback to interpolation is deactivated by default. To enable the fallback functionality, following configuration option has to be set:

```
cpa.predicate.refinement.newtonrefinement.fallback = true
```

5. Evaluation

The implementation of *Newton Refinement*[1] was tested on the SV-COMP 2018 benchmark set[2] consisting of a huge number of different programs. The set is commonly used to perform benchmarks of software verification tools. The benchmark itself was performed using *BenchExec*[7], a benchmark framework developed to evaluate the effectiveness and efficiency of software verification tools. The benchmarks where executed on Linux[3] machines with Intel Xeon CPUs[4]. Each run was limited to use at most 15GB of RAM and 2 CPU-cores. The time limit was set to 600s per run.

The evaluation of our implementation of Newton Refinement is split into three sections. In Sect. 5.1 we test several configurations of Newton Refinement against each other to find the best configurations. In Sect. 5.2 we select one configuration of Newton Refinement and compare the different SMT solvers supported by CPAchecker. Moreover, we will analyze the influence of solver-based quantifier elimination for those solvers that support quantifier elimination. Finally in Sect. 5.3 we compare Newton Refinement with Craig Interpolation.

Another aspect evaluated is the implementation of quantifier elimination, which we present in Sect. 4.2. The different implemented methods were evaluated regarding the number of successful quantifier eliminations over the benchmark set.

5.1 Comparing the Configuration Options of Newton Refinement

In this section we compare the evaluation results of CPAcheckers predicate analysis with Newton Refinement for several configuration combinations. We

[1] Based on CPAchecker revision *trunk:29456*
[2] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp18
[3] Linux 4.15.0-38-generic
[4] Intel Xeon E3-1230 v5 @ 3.40 GHz

compare these configurations regarding their effectiveness, respectively their ability to successfully prove or disprove programs of the benchmark set. For all tests in this section, we will use *MathSAT5*[5] as SMT-solver. We use MathSAT5 because it is the default solver of CPAchecker and most used in other analysis configurations. We test the Newton Refinement configurations for two types of adjustable-block encoding. Adjustable-block encoding can be configured in several ways to set the block size of the blocks between abstraction states. For the first type we choose a block size of 1, which means each edge of the CFA acts as a block. For simplicity, we will refer to this configuration as single-block encoding(SBE). The second type we use is the default configuration of CPAchecker, where only loop heads act as abstraction states. We will refer to this configuration as large-block encoding(LBE).

The configurations of Newton Refinement are named as followed. "Block" and "Edge" refer to the abstraction level used by Newton Refinement(cf. Sect. 4.1.1 and 4.1.2). The additional "IC" states that infeasible-core abstraction was used (cf. Sect. 4.1.3). When the configuration name contains "LV", we quantify variables which are not future live (cf. Sect. 4.1.4). If the configuration name contains an additional "fb" the fallback to interpolation(cf. Sect. 4.3) option is turned on.

For each configuration we present the total number of correct/incorrect proofs(true) and alarms(false) as well as the number of timeouts and other errors. We present the benchmark results of the configurations tested in this section in Tab. 5.1. The first group shows the results of the predicate analysis with Newton Refinement for a configurations where the ABE block size is set to 1(SBE). The second group presents results where the default ABE setting was used (LBE). The last group contains configurations of the predicate analysis, where interpolation was used. It contains the default configuration of predicate analysis as well as configurations of Newton Refinement, where the fallback to interpolation is activated. This group is included to evaluate the fallback option of Newton Refinement at the end of this section. Moreover the inclusion of interpolation-based predicate analysis allows a better estimation of how effective the Newton Refinement configurations are. A more detailed comparison of Newton Refinement with interpolation will follow in Sect. 5.3. We highlight the best configuration for each group and result type with a green background.

[5]http://mathsat.fbk.eu/

Table 5.1: CPAchecker benchmark results for different configurations of Newton Refinement using Mathsat5 as solver.

	Correct true	Correct false	Incorrect true	Incorrect false	Timeout	Other Error
SBE-Edge	1329	266	0	3	3166	659
SBE-Edge-IC	1419	241	0	1	1918	1844
SBE-Edge-LV	1287	283	0	2	3242	609
SBE-Edge-IC-LV	1383	222	0	2	1970	1848
SBE-Block	1419	281	0	1	2872	850
SBE-Block-IC	1420	281	0	1	2880	841
SBE-Block-LV	1398	279	0	1	2919	826
SBE-Block-IC-LV	1399	277	0	1	2917	829
LBE-Edge	1786	598	0	3	2473	563
LBE-Edge-IC	1884	628	0	1	1845	1065
LBE-Edge-LV	1894	686	0	3	2241	599
LBE-Edge-IC-LV	1877	625	0	1	1857	1063
LBE-Block	1538	412	0	1	2685	787
LBE-Block-IC	1539	411	0	1	2680	792
LBE-Block-LV	1545	411	0	1	2664	802
LBE-Block-IC-LV	1546	410	0	1	2666	800
SBE-Interpolation	1514	324	0	1	2565	1019
SBE-Block-IC-LV-fb	1405	283	0	1	2951	783
SBE-Edge-IC-LV-fb	1472	295	0	1	2683	972
LBE-Interpolation	2055	731	1	3	1880	753
LBE-Block-IC-LV-fb	1620	435	0	1	2732	635
LBE-Edge-IC-LV-fb	1981	711	1	1	1985	744

First, we will evaluate the results of configurations using ABE with block size 1(SBE). Therefore, we look at the the first group. Overall the configurations using block-level refinement seem to perform better. Only "SBE-Edge-IC" achieves a similar number of correct proves, yet it finds noticeably less correct false results. When we look at the infeasible-core abstraction, it positively influences both edge-level and block-level abstractions for true proofs. For "SBE-Edge-IC(-LV)", infeasible-core abstraction reduces the number of correct alarms and increases the number of errors. These errors are mostly cases where the created sequence of assertions is not strong enough and the error path is discovered a second time. Live variable abstraction has a negative effect on the number of correct proofs for both edge-level and block-level abstraction. "SBE-Edge-LV" is also the configuration finding the most correct falses.
The second group of Tab. 5.1 contains the configurations of Newton Refinement where the default configuration of ABE was used. First we notice that all configurations using LBE are better than even the best configuration using SBE. The best LBE configuration("LBE-Edge-LV") correctly verifies 50% more programs than the best SBE configuration("SBE-Block-IC"). This shows that Newton Refinement profits a lot from the more abstract model introduced by larger block sizes. When we look into the differences between configurations using block-level abstractions and those using edge-level abstractions, we can observe two things.

The first observation is that edge-level abstraction creates more correct proofs and alarms. The difference is quite big: the best edge-level result "LBE-Edge-LV" is successful in 2580 cases, while "LBE-Block-IC-LV"(the best block-level result) solves only 1956 cases.

The second observation is the effect of the optimizations "IV" and "LV" on the results. When we look at the edge-level abstraction "LBE-Edge-IC", it successfully verifies 128 more programs than "LBE-Edge". As for the SBE cases, infeasible-core abstraction results in a higher number of errors. The reason for these errors are state assertions that are not strong enough to prove the paths infeasiblity.

The live-variable abstraction has an even bigger effect on the number of successful verifications. The configuration "LBE-Edge-LV" produces 198 correct results more than "LBE-Edge". The combination "LBE-Edge-IC-LV" performs worse than both "LBE-Edge-LV" and "LBE-Edge-IC" but is still better than "LBE-Edge".

Block-level abstraction does not profit as much from the optimizations. The best block-level configuration "LBE-Block-IC-LV" creates just 6 successful results more than "LBE-Block". One possible explanation for this small effect may be the complexity of the formulas because LBE encodes several state-

ments in one formula. For complex formulas, the simple methods of quantifier elimination introduced in 4.2 are less likely to succeed. The number of errors is roughly the same for all configurations using block-level abstraction,too.

Finally we look at the third group of configurations in Tab. 5.1. Regarding the influence of SBE and LBE we can make the same observations as for the first two groups. The LBE configurations outperform all SBE configurations. The fallback to interpolation configurations are, as expected, better than the same configurations without fallback. But they do not reach the same amount of correct results as interpolation does. The comparison of interpolation with Newton Refinement will follow in Sect. 5.3.

Based on the findings of this section we can sum up some observations. First, Newton Refinement profits from the big block sizes created by the ABE default settings. Second, for LBE the edge-level abstraction outperforms the block-level abstraction with up to 25% more correct results. Third, "LBE-Edge-LV" is the best overall configuration and we will use it in all following benchmarks.

5.2 Evaluation of Newton Refinement regarding Different Solvers

In this chapter we will analyze the effect of different solvers on the results of the predicate analysis with Newton Refinement. We want to analyze which solver is most successful when using Newton Refinement. Another aspect we want to analyze is the influence of solver-based quantifier elimination for solvers supporting quantifier elimination.

We will perform benchmarks using *MathSAT5*[6], *Z3*[7], *SMTInterpol*[8] and *Princess*[9] as SMT solvers. Each solver has different abilities. MathSAT5 and Z3 are the only solvers that support the theory of Bitvectors and can compute bitprecise SMT formulas. These two are also the only solvers supporting floating point formulas. Moreover, only Z3 and Princess support quantifier elimination.

We present the results of CPAchecker predicate analysis runs with the different solvers in Tab. 5.2. All runs presented in this table are configured to use Newton Refinement with edge-level abstraction and activated live variable abstractions(cf. "LBE-Edge-LV" in Sect. 5.1). For all runs, ABE is configured to end block formulas only at loop heads.

[6]http://mathsat.fbk.eu/
[7]https://github.com/Z3Prover/z3
[8]https://ultimate.informatik.uni-freiburg.de/smtinterpol/
[9]http://www.philipp.ruemmer.org/princess.shtml

Table 5.2: CPAchecker benchmark results of Newton Refinement ("LBE-Edge-LV" configuration) for different solvers.

	Correct true	Correct false	Incorrect true	Incorrect false	Unknown
MathSAT5-none-bv	1894	685	0	3	2841
Z3-none-bv	1574	389	0	10	3450
Z3-light-bv	1576	390	0	10	3447
Z3-full-bv	1366	340	0	9	3708
MathSAT5-none-int	1807	765	12	237	2602
Z3-none-int	1576	554	8	223	3062
Z3-light-int	1575	554	8	223	3063
Z3-full-int	1430	493	8	217	3275
Princess-none-int	1390	251	8	175	3599
Princess-light-int	1393	250	8	175	3597
Princess-full-int	1224	280	8	171	3740
SMTInterpol-none-int	1500	400	11	143	3369

In order to provide a better comparability between all solvers we split the benchmark configurations in two groups. The first group performs bitprecise computations and only contains benchmarks for MathSAT5 and Z3. The second group contains all solvers and is configured to encode bitvectors as integers. This is a configuration supported by all solvers. Moreover, in this group, floats are encoded as rationals with exception of Princess, which only supports the encoding of floats as integers. For the default stack size Z3 often stops with segmentation faults. To avoid these errors, we set the stack size for these runs to 1 GB.

The configurations are named starting with the solver name. The next name component is "none","light" or "full" and specifies the setting of the solver-based quantifier elimination. As only Z3 and Princess support quantifier elimination, the other solvers are only tested with "none". The last component is either "bv" for bitvector encoding or "int" for integer encoding.

First we will look at the solvers supporting the theory of bitvectors, MathSAT5 and Z3. The configurations using bitvector encoding are presented in the first group of Tab. 5.2. As we can easily see, using MathSAT5 as a solver produces the best result. But Newton Refinement also produces a good number of correct proofs and alarms when using Z3. Of the 3 configurations of Z3, we find the best result for "Z3-light-bv". This is the configuration where Z3 solves quantifier eliminations with a best effort strategy if they could not priorly be solved by DER and UPD(cf. Sect. 4.2). The result of "Z3-none-bv" is only slightly worse than the "light" setting. Interestingly, the weakest result for

Z3 is the configuration using the "full" quantifier elimination strategy. In this case, the quantifier elimination often leads to timeouts or segmentation faults, even with a stack size of 1 GB.

For the configurations using integers to encode bitvectors we notice a far higher number of incorrect results. This is not surprising as the integer encoding is not bitprecise. The high number of incorrect results is an issue of all solvers. Moreover, we notice that MathSAT5 and Z3 find more correct alarms as for bitvector encoding. For the configurations using Princess or SMTInterpol, the number of correct results is smaller than for those using MathSAT5 or Z3. The solver-based quantifier elimination has only a small effect for a "light" strategy and a negative effect for a "full" strategy. This applies for both supporting solvers, Z3 and Princess. The full quantifier elimination seems to add too many complex computations to be useful.

Based on the results of this section we conclude that using bitvector encoding is much more precise. Therefore, we will concentrate our further efforts on the solvers supporting bitvectors(MathSAT5 and Z3). Regarding the quantifier elimination we will use the "light" configuration.

5.3 Comparing Newton Refinement with Craig Interpolation

In this section, we compare Newton Refinement and Craig Interpolation regarding both their effectiveness and efficiency. The goal is to evaluate in how far Newton Refinement can be regarded as an alternative to Craig Interpolation. Another interesting observation will be if Newton Refinement can solve cases where interpolation fails.

In order to get more relevant results, we use a second benchmark set consisting only of programs which need at least one refinement step. For Newton Refinement we use the configuration "LBE-Edge-LV" of Sect. 5.1. This means we use large-block encoding, edge-level abstraction, and live variables. The interpolation is performed with the default configuration for the predicate analysis of CPAchecker. Moreover we will use the two solver configurations "MathSAT5-none-bv" and "Z3-light-bv" which are the most promising solver configurations(cf. Sect. 5.2). We define four test cases, "MathSAT5-Interpolation", "MathSAT5-Newton", "Z3-Interpolation" and "Z3-Newton".

In Tab. 5.3 we present the effectiveness of the defined Configurations. The results are divided in two groups. The configurations of the first groups use MathSAT5 as a solver while the second uses Z3. In each group we add an

Table 5.3: CPAchecker benchmark results of Newton Refinement and Craig Interpolation using MathSAT5 and Z3 for a reduced benchmark set

	Correct true	Correct false	Incorrect true	Incorrect false	Unknown
MathSAT5-Interpolation	1672	735	1	3	2184
MathSAT5-Newton	1507	684	0	3	2401
subset	1427	666	0	1	2084
Z3-Interpolation	1270	391	0	10	2924
Z3-Newton	1188	391	0	10	3006
subset	1119	355	0	10	2819

additional row called subset. It represents the subset of programs where both interpolation and Newton create the same result.

When we look at the first group we can see that interpolation produces 216 correct results more than Newton Refinement. Regarding the incorrect results, interpolation produces one incorrect proof and Newton none. By comparing the configurations with the subset of solved programs we can determine how many programs can be solved by only one configuration. Interpolation manages to compute 245 correct proofs and 69 correct alarms, which could not be solved by Newton. In return, Newton is able to solve 80 correct proofs and 18 correct alarms where interpolation failed. This shows that Newton Refinement can verify a reasonable amount of problems for which interpolation fails or is to slow. When we inspect the cases where interpolation fails and Newton succeeds, we observe that in many cases interpolation ends with an error. The error states that interpolation is not applicable for the specific problem. For the cases where interpolation succeeds and Newton fails, the reason often is a timeout. Thus, Newton would probably solve these cases for a higher timelimit or additional computing resources.

A similar observation can be made for the second group where Z3 is used. Interpolation can create 82 correct proofs more than Newton. The difference between the two configurations is smaller than for MathSAT5. Next we regard the cases exclusively solved by only one configuration. The number of cases exclusively solved by interpolation include 151 correct proofs and 46 correct alarms. Newton manages to create 69 correct proofs and 46 correct alarms of programs unsolved by interpolation. As for MathSAT5, this shows that Newton Refinement can increase the number of solvable programs.

The performance indicator we analyze is the CPU time used to create a proof or alarm. Figures 5.1 and 5.2 contain the scatter plots of the CPU time used for generating correct results. The horizontal axis indicates the time needed by Craig Interpolation while the vertical axis shows the time needed

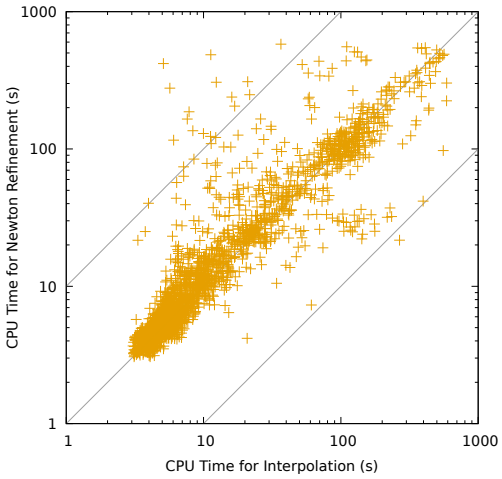

Figure 5.1: Scatter plot Newton Refinement vs. interpolation using MathSAT5

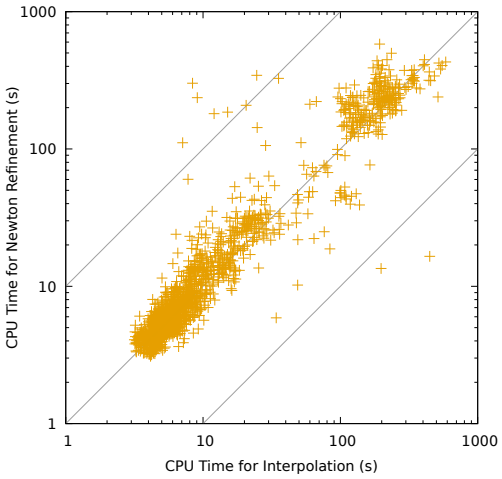

Figure 5.2: Scatter plot Newton Refinement vs. interpolation using Z3 with light quantifier elimination

by Newton Refinement. The central diagonal line marks the area where both configurations need the same amount of time. It also divides the plots in two halfs, the top left half containing results where interpolation was faster than Newton Refinement and the bottom right half where interpolation was slower.

Based on the plot in Fig. 5.1 we can observe two kind of patterns. The first observation is the cluster along the diagonal line. It indicates that for a good part of the test cases both interpolation and Newton Refinement need a similar amount of time. The second observation are the data points which are more distant to the diagonal. We can see that the number of points in the top left half is bigger than in the bottom right half. This means that the amount of programs where interpolation outperforms Newton Refinement is greater than the other way around. But the fact that not all points are in the top left half of the plot also means that in some cases Newton Refinement performs better.

In Fig. 5.2 we present the same kind of plot using Z3 as solver. As for MathSAT5 also for Z3 we notice that the majority of the points can be found along the diagonal. But for Z3 we see two clusters along the diagonal. One cluster of programs that can be solved rather quickly in the bottom left and another cluster in the top right. The rest of the data points are more evenly distributed. This means both Newton-based and interpolation-based predicate analysis solve some problems faster. The amount of these points is also roughly equal. So we can determine that the difference regarding the solving time between Newton and Craig is rather small, when we use Z3.

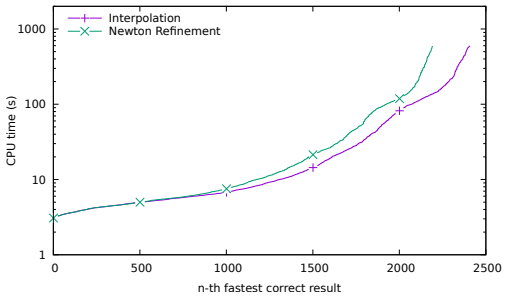

Figure 5.3: Quantile plot for MathSAT5

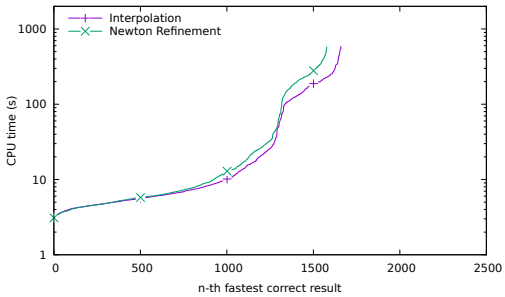

Figure 5.4: Quantile plot for Z3-light

In Fig. 5.3 and Fig. 5.4, we display the quantile plots for the configurations of Tab. 5.3. When we look at the left figure, we can see the plot for MathSAT5. Interpolation-based refinement can solve more programs than Newton. The form of the curves is rather similar, which supports our observation in the scatter plot that for a big number of programs interpolation and Newton need a similar amount of time. The difference of solved problems in a specific time grows continously. This indicates that the advantage of interpolation applies for programs of all complexities.

In Fig. 5.4 we can see the quantile plot of the predicate analysis using Z3 as a solver. As we could already establish, Z3 performs worse than MathSAT5. The two curves share a similar form. Both configurations perform very similar for the fastest 1300 verification problems, which can all be solved within a time of roughly 50s. After the fastest 1300 cases the required time for solving increases fast. For both configurations, there are only few cases that can be solved in 50s - 100s. Yet both configurations can solve several hundred cases where the computation takes more than 100s. For these cases, the difference between interpolation and Newton becomes bigger. Interpolation can solve more such programs than Newton.

These observations correlate with the two clusters of the scatter plot(cf. Fig. 5.2). The bottom left cluster contains cases with an execution time of less than 50s. The top right cluster contains the cases where the execution takes longer than 100s.

In conclusion we can note that interpolation based predicate analysis performs better, especially when using MathSAT5. Yet also Newton Refinement can provide some advantages and is faster for some programs. Moreover, Newton Refinement succeeds in many cases where interpolation is not applicable. For these cases, Newton Refinement adds a new way of verification.

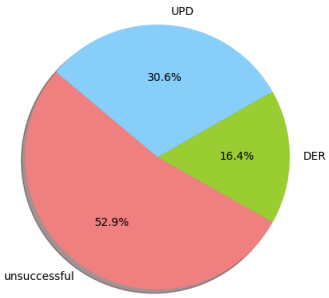

Figure 5.5: Successful quantifier eliminations MathSAT5

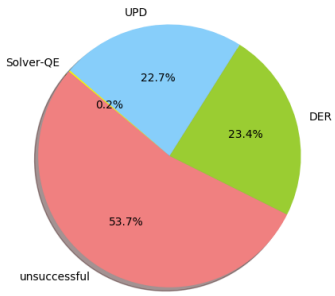

Figure 5.6: Successful quantifier eliminations Z3-light

5.4 Evaluation of Quantifier Elimination

In order to evaluate the effect of the implemented methods to perform some best effort quantifier elimination, we will analyze which method is responsible for successful quantifier eliminations. The analysis is based on the statistics for quantifier elimination over the whole set of benchmark programs used in Sect. 5.3. We analyze the two configurations of Newton Refinement using MathSAT5 and Z3 as solvers.

In Fig. 5.5 and Fig. 5.6, we present the percentages of successful and unsuccessful quantifier eliminations. The successful quantifier eliminations are grouped by the method responsible for the elimination. Both charts are based on the whole set of programs used in Sect. 5.3. The order of the performed operations is as followed. First, we try DER then UPD and when both were unsuccessful, we apply the light strategy of quantifier elimination for Z3. Another order would probably lead to other results because the light strategy also makes use of the two other methods.

In Fig. 5.5, MathSAT5 was used as solver. The percentages are based on $\approx$ 105 million quantifier eliminations performed while proofing the set of programs. With around 30.6% solved quantifier eliminations, Unconnected Parameter Drop is the most useful of the methods that we implemented. Together with Destructive Equality Resolution we were able to perform half of the quantifier eliminations without the need for a supporting solver.

On Fig. 5.6, we present the statistics for the quantifier eliminations using the Z3 solver on the same set of programs. We used the light quantifier elimination strategy as it performed far better than full quantifier elimination in the previous benchmarks in Sect. 5.2. Surprisingly, the integrated quantifier elim-

ination does not outperform the rather simple methods DER and UPD. Only a small fraction of additional quantifier eliminations could be solved using the solver. For Z3 DER is more successful than UPD.

6. Conclusion

Over the course of this thesis we presented the concept of CEGAR, introduced Newton Refinement and implemented it in CPAchecker. Moreover we evaluated various configurations of CPAcheckers predicate analysis using Newton Refinement. In this context we also analyzed which solvers are the best option to be used with Newton Refinement. Finally we compared our work with the already implemented Craig Interpolation.

Even though Newton Refinement cannot create as many correct results as Craig Interpolation, it is interesting for several reasons. Firstly Newton Refinement is able to solve some problems where interpolation does not succeed. This is interesting as it extends the number of programs that can be verified. When a problem cannot be solved using Craig Interpolation, Newton Refinement presents a second configuration to try.

Secondly Newton Refinement is independent of the availability of an interpolating solver. Thus it can be used with solvers where interpolation is no option. Z3, one of the solvers supported by CPAchecker, will no longer support interpolation in future releases[1]. This makes MathSAT5 the only solver supported by CPAchecker that supports both interpolation and the theory of bitvectors. In order to stay independent of a single solver it is vital to have alternatives like the presented Newton Refinement.

A possible future extension of CPAchecker could be to implement a routine to try Newton refinement if interpolation fails. A similar configuration "fallback to interpolation"(cf. Sect. 4.3) has been implemented in this thesis. It tries to perform an interpolation refinement when Newton fails.

Even though we present extensive benchmarks within this thesis, further configurations of predicate analysis using Newton Refinement are possible. Therefore additional benchmarks using other configurations may be interesting.

[1] https://github.com/Z3Prover/z3/pull/1646

Bibliography

[1] Tom Ball and Sriram Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical report, 2002.

[2] Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 9636, pages 887–904. Springer, 2016.

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

[4] Dirk Beyer and Karlheinz Friedberger. Domain-independent multi-threaded software model checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 634–644. ACM, 2018.

[5] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, LNCS 6806, pages 184–190. Springer-Verlag, 2011.

[6] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design*, pages 189–197. FMCAD, 2010.

[7] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer (STTT)*, 2017.

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

[9] William Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[10] 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*, ESEC/FSE 2017, pages 487–497, 2017.

[11] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *Computer Aided Verification*, pages 72–83. Springer, 1997.

[12] Kenneth L. McMillan. *Interpolation and Model Checking*, pages 421–446. Springer, 2018.