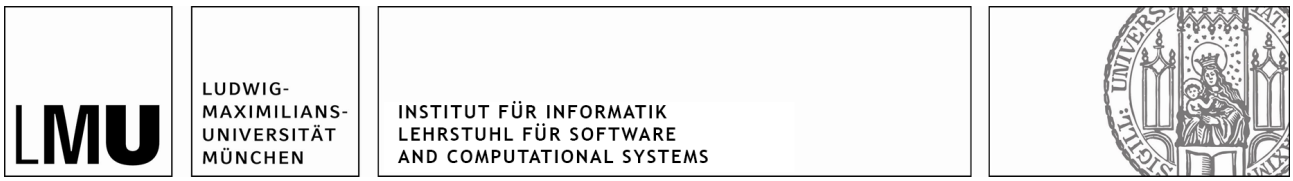


Master Thesis
in Computer Science

New Approaches and Visualization for Verification Coverage

Maximilian Hailer

Supervisor:	Prof. Dr. Dirk Beyer
Mentor:	Dr. Philipp Wendler
Submission Date:	11.06.2022

Abstract

Formal verification can help prove the correctness of software systems. Multiple results are possible when verifying software with typical model checkers: true, false, or unknown. While comprehensive statistics exist for completed analysis, cases like timeout or out of memory provide little information about the fulfillment of the specified properties.

Therefore, we propose and implement new software-verification coverage measures to help understand how much of the code is already considered by the verifier. In contrast to the well-known test coverage, verification coverage does not have a common definition yet. Consequently, we try to clarify this with a literature overview to discuss the different interpretations depending on the research area. Afterward, we define verification coverage as a measure for the verifier to depict the current progress regarding the coverage of the program. As a second scenario we could have the case that we want to verify a program, but we use an inappropriate specification. Consequently, the result is not meaningful, since the verifier did not check the properties we actually wanted to verify. Having a proper coverage measure can help detecting cases like this by indicating parts of the program which were not covered due to the specification. What coverage in this context means can depend on the used verification analysis and will be discussed within this thesis.

Different visualizations augment theses approaches by granting the user an alternative view of the coverage of the program. This can be accomplished by coloring or using heat maps for locations of a control flow automaton (CFA) or source code lines. Furthermore, it is also interesting to know how much the verification tool has covered during the analysis at a certain time to determine the overall progress. Therefore we suggest an approach for visualizing the development of the verification coverage over time. All mentioned ideas packaged together should enhance the usability of verification tools so that users can determine the progress of the analysis on unknown cases and on the other hand can check if the used specification lead to a high coverage if the analysis finished without problems.

Acknowledgments

My special thanks go to my mentor Dr. Philipp Wendler, who gave me great and fast feedback on my drafts. Our weekly discussions resulting in constructive advice helped me improve and rethink concepts. I also want to thank my supervisor Prof. Dr. Dirk Beyer, for giving me the opportunity to write this thesis. Beyond that, many thanks go to my mother, who always believed in me and helped me to have the time to focus on my work.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Nowadays, our society relies heavily on software. It can help simplifying and automating processes. This often leads to a more efficient workflow and opens use cases that would not be considered without software. We have a good chance to meet systems that rely on software in our everyday life. Sometimes they are in the background and assist the demanded process. This could be the case for software which coordinates trains or deals with withdrawals from ATMs. However, there are also more obvious use cases like email client programs on computers to send messages or smartphone applications to make video calls or search the web.

When dealing with software, it is crucial to keep in mind that there is a particular possibility of failure in using a software system. Software failure does not mean wrong user inputs, which the program does not accept. Instead, we specify failure here as unexpected behavior of the program, meaning that the program is not working according to its specifications. This could typically be a program crash. Those incidents most often have one of two reasons. One is the case when we have wrongly written software code, so-called bugs. The second case is due to hardware problems. Some can be nondeterministic, like random bitflip occurrences. Some others are caused by a bad hardware architecture or internal component faults, which directly affect the software and lead to unexpected behavior. Since the second reason is focused on a hardware-specific research field, we want to limit our scope to software bugs.

For non-safety-critical systems, having bugs in software can cause some

annoyance for the user, since they make it more challenging to use the system as designed. Usually, they get reported and fixed afterward via updates. Nevertheless, when dealing with safety-critical systems, like software for airplanes and rockets or self-driving cars, it is absolutely important to prevent the occurrence of critical bugs. Otherwise, we could have outcomes like death, injury, and severe damage to properties or the environment. The consequence of this is that we should know before deploying productive systems that the software is safe.

1.2 Formal Verification

To accomplish safety checks, we need a specification to know when we consider software safe. When we have one, we can check if the software fulfills this specification. There are typically two main approaches for this process: testing and formal verification. While testing focuses on checking the program for specific use cases, formal verification has the aspiration of verifying all possible program paths. This comes with the cost of high demand regarding computational power since checking all program paths can lead to a state explosion with an exponential runtime or memory complexity. This opens a research field to find efficient algorithms which have better performance in checking the whole program than just naive iterating through every possible path. Ideally, when the analysis is done, we have one of two different outcomes; either the program fulfills the given specification, or we have found a violation. When something during the analysis goes wrong, there is a third possible result, which we call unknown. Reasons for this can be insufficient memory, taking too much compute-time, or occurrences of bugs and errors within the analysis software itself. The problem with an unfinished analysis is that we are not sure about fulfilling the given specifications. Therefore we cannot approve the program safe. The point here is that an unknown case does not help the verification engineer with its work. Consequently, the software developers of the verification analysis tool need to optimize their verifier based on those unknown cases to lead to correct results. But even if the outcome is not unknown we need to assure that our used specification is correctly defined. If we want to verify that our program does not reach certain parts of code, but our specification does not consider all of these parts as property violation our verifier will not detect it. This would lead to correct true result for the given specification, but for our actual problem the verdict would be nevertheless incorrect. Checking for a right a specification is not a trivial problem, since it depends on the user what they actually assume as correct code.

Taking the relevance and importance of efficient and reliable software into

account and considering that verifying software is compute-intensive and can lead to unknown results, it would be helpful to have as much detailed information about the uncompleted analysis as possible. Similar approaches like conditional model checking [4] exists, but in this thesis we want to focus on a measure or visualization which describes the coverage of the program. In addition, it would be beneficial if we could check how much the used specification affected the verifier in analyzing the program. Hence, we could then determine how appropriate the used specification for our problem was. Therefore, we propose in this thesis multiple verification coverage measures and visualizations to deliver more information helping to compensate the mentioned issues.

1.3 Comparison to Software Testing

In contrast to software testing, no common verification coverage definition exists. We will see within the literature overview chapter that this kind of coverage depends on the actual application environment. When working with test coverage, we actually run the program and look at lines that were executed by tests or not. When we want to adapt this idea to verification coverage we need to consider the issue that we are actually not running the program; instead we analyze it depending on our given specification. Therefore we need a replacement of the criteria "line x executed". The obvious answer would be we consider the alternative "line x analyzed". Consequently, we then need to answer the question what does "analyzed" mean in this context? There is no definite answer to this question, wherefore we try to find suitable new approaches to calculate the verification coverage.

1.4 Contribution

In this thesis, we elaborate on methods that should help the verification engineer to better understand how much of the given program was already processed and covered by the verifier. In the beginning we give a literature overview about the current research regarding verification coverage. Afterward, we clarify all needed background knowledge for the upcoming chapters. We separate between our proposed ideas and the concrete implementation procedure. Regarding the main topic of verification coverage, we focus our approaches on the concept of *completeness of properties* and *completeness of verification procedure*. Especially for the last concept we mainly work with visualization techniques, since it should be better comprehensible for the verification engineer to interpret graphs and colorful pictures instead of raw numbers. This is due to the high architectural complexity of current verification

frameworks, where in contrast to just running a test suite, there are no reliable dedicated progression numbers available.

We implement our work in the verification framework CPAchecker. There already exists an approach of printing a GCOV-formatted verification coverage file. It can be used to create an HTML report which shows the coverage per line for the source code. This approach has the disadvantage that it is not integrated within the default HTML report of CPAchecker and is limited to the source code statements regarding the coverage visualization. Therefore we implement our own visualization approaches in CPAchecker and enhance the resulting `Report.html` so that the user automatically has access to the graphics without the need to run extra commands. In addition, we do not limit our scope on source code, instead we also consider further verification-related structures to visualize. We also refactor and enhance the current code regarding verification coverage in CPAchecker to allow, due to a more modular design, a simple way to add further verification coverage measures. Those can be then automatically visualized in the HTML report without the need of implementing a dedicated visualization method.

In the end we evaluate our approaches on different verification analyses and programs. We show the differences and discuss the advantages and disadvantages of each proposed verification coverage idea.

2 Literature Overview

Verification coverage is a term in the scientific literature used in different research areas. When trying to cluster all the contributions, two main application fields result. First, hardware-design verification, and second, software model checking. However, the concepts and ideas behind this topic are not strictly separated from the mentioned application fields. Therefore we can say that these clusters are not disjoint. In this thesis, we will focus our new approaches on software verification frameworks like CPAchecker, which belongs to the software model checking application area. Nevertheless, we will discuss in this chapter both fields to give an overview about the current state of research in the literature about verification coverage.

2.1 Verification Coverage in Hardware-Design Verification

According to William K. Lam, "hardware design verification encompasses many areas, such as functional verification, timing verification, layout verification, and electrical verification" [20]. The book of the mentioned source is mainly about "functional verification" and refers to it as "design verification". We follow the convention of this book since the term "verification coverage" is most commonly used in this area. On the other hand, Andrew Piziali claims that "functional verification is demonstrating the intent of a design is preserved in its implementation" [22]. This means that the objective is to bring the design intent, the specification, and the implementation into coincidence. When considering the definition of William K. Lam this becomes

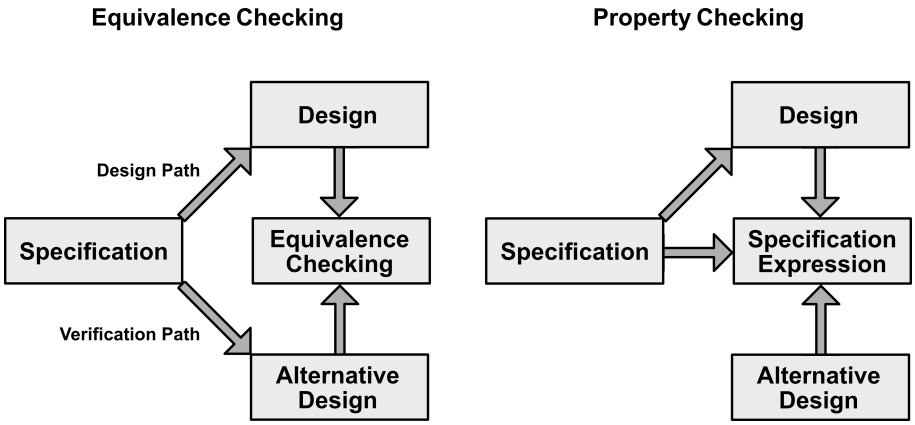


Figure 2.1: The basic principle of equivalence checking (left) and property checking (right) [20].

clear in combination with the figure about the basic principle of design verification (cf. Fig. 2.1). There are two approaches of design verification depicted in Fig. 2.1. Property checking (right) has a design based on the given specifications. Furthermore the design has to satisfy the specification expressions, meaning that the verification engineer is doing a transformation from specifications to an implementation using properties deduced from the specifications. This methodology is typically adapted in model checking [20].

When looking at equivalence checking (left figure), we see that the verification engineering team tries to build an alternative design with the same specification from the design verification team and then checks for equivalence. The methodology in this kind of verification typically has a test plan. The book claims that since, in reality, it is infeasible to uncover all bugs by verifying a set of specifications thoroughly, a measure of verification quality is desirable to track the progress of the verification. Commonly used ones are code coverage and functional coverage. Code coverage is defined similarly as in software testing. It measures the percentage of code stimulated (tested), whereas functional coverage is mainly used in hardware verification. Its use case is to approximate the percentage of functionality verified [20].

In the following, we want to discuss more profoundly functional coverage. Since this measure varies depending on the kind of verification, we look into the two main categories of design verification: Simulation-based and formal method-based verification. There are also hybrid forms called semi-formal verification, which combine both positive aspects of each category: Exhaustive regarding inspecting the state space like in formal verification and good scalability and ease to use like in simulation-based verification.

2.1.1 Verification Coverage in Formal Method-based Verification

Formal-based verification uses mathematical methods to verify that a design works correctly for all allowed inputs [19].

There are two types of verification techniques: reachability analysis and deductive methods. The first technique includes approaches like model checkers. This means a formal mathematical specification, such as temporal properties, is needed to define which states of the program are allowed to be reachable and which are not. The verifying tool tries to exhaust all possibilities to conclude: Properties violated or not. As already described in the previous chapter, there is also the option that the verifier does not come up with a conclusion, and the result then would be unknown. As we will show in this thesis, a coverage measure can help get more information when dealing with unknown results. In the context of hardware verification, the model checking properties, which are defined in temporal logic, describe parts of the circuit's behavior. There are approaches to estimate the functional coverage in bounded model checking. An application case can be where a possible verification engineer wants to know if the proven properties describe the complete functional behavior of the circuit [13]. The goal of the coverage definition here is to automatically detect scenarios where none of the properties specify the value of the considered output. The author describes a way to generate a coverage property for each considered output to achieve this. The next step is to check if the coverage property holds. If true, then the output value is only determined by the properties. In this scenario, a higher coverage means more coverage properties were found.

2.1.2 Verification Coverage in Simulation-based Verification

Simulation-based verification is the most frequently used technique for complex designs. Whereas this approach is limited to a test bench, formal methods consider a design exhaustively, but this comes with a cost of high computational effort [24, 20].

The test bench consists of code that supports the operations of the design. It is then possible to apply the so-called input stimuli, a kind of test vector, to the test bench. As a result, we get an output which we can compare to the reference output. In contrast to conventional testing, the input vector can be generated during the simulation. Similarly, the reference output can be generated in advance or on the fly [20]. Coverage in this approach plays an important role. Since the coverage measure is directly involved in the stimulus

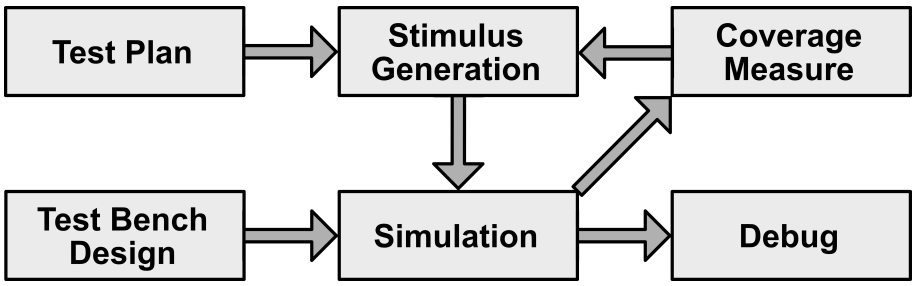


Figure 2.2: Flow of simulation-based verification [20].

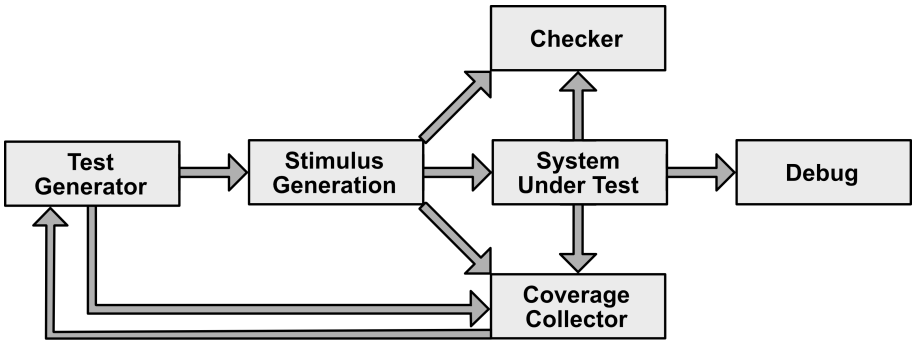


Figure 2.3: Flow of coverage-driven verification [1].

generation, it is important to deduce the relevant information appropriately during the simulation (cf. Fig. 2.2). The abstract definition here is how much the design is stimulated and verified during simulation. A coverage tool can be used to measure the code coverage or functional coverage to accomplish this. While the first one is analogously defined as code coverage in software testing, the latter is the percentage of exercised functionality. In addition to automatically generating stimuli, the test designer can manually create additional tests to increase the coverage potentially, or they can leave out tests that do not increase coverage to minimize redundancy [20].

Based on simulation-based verification a subarea has evolved called coverage-driven verification (CDV). This approach allows in addition to the automatic stimuli generation, also the generation of effective tests to explore a system under test (SUT) to achieve steadily higher coverage.

When we compare Fig. 2.3 to the one before, we can see that the coverage metrics are now applied to the test generation directly instead of just to the stimuli generation. This has as a consequence, that the coverage now influences the generation of the next tests. Automatically generating tests needs

much information. For that reason, we can see in Fig. 2.3 that the Coverage Collector depends on many other modules. The procedure applied to the gathered information is called coverage closure. It aims to identify coverage holes and create new tests to cover those [1]. The resulting feedback loop of coverage analysis to test generation is called coverage-directed test generation (CDG). In practice, CDG is a difficult challenge, wherefore there are research attempts based on machine learning to accomplish this process [17].

There are also coverage measures defined on the hardware itself. They belong to the group of circuit coverage. The idea behind this group is to identify the covered physical parts of the circuit [10]. Advantages are that realizing the measuring process is usually straightforward, and also, the interpretation of the result is often no problem. On the other hand, it is not easy to generate new tests based on this data for the simulation-based approach. Some popular examples for circuit coverage measures are latch and toggle [15, 18]. A latch is covered if it is changing its value at least one time, whereas a toggle is covered when it is changing at least twice, meaning that the bit of the register is changed, for example, from value 0 to 1 and then back from 1 to 0 during simulation. All coverage measures before can be categorized into syntactic coverage measures since they make use of specific formalism to describe how much of a design is covered [10]. To close the spectrum of coverage categories in this section, we also need to mention the so-called semantic-coverage measures. They require interaction with the user to measure the part of the functionality of a design depending on the input. Commonly used ones are assertion coverage, which measures the percentage of covered assertions. The user defines an assertion as a condition that needs to hold during the simulation. Some more advanced approaches would be mutation coverage, where the user slightly changes (mutates) the design and checks if this leads to any bugs. Here the coverage would be measured as the relation of failed tests to all test runs, where we keep the same test and modify the design. This would lead to multiple coverage values since each test is considered separately.

2.2 Verification Coverage in Software Model Checking

Software model checking belongs to the group of formal verification. Therefore there are some similarities to Chapter 2.1.1. In contrast to the application of verification coverage within the hardware domain, we look in this chapter specifically at software model checking. The main difference here is that we do not look at the design of the hardware. This leads to alternative coverage definitions discussed in the following sections.

2.2.1 Completeness of Properties

Discussing the completeness of properties is essential since even though model checking is an exhaustive method, it is nevertheless possible that bugs can occur. The verifier cannot detect erroneous behavior if it does not violate the specification. Therefore we can only conclude that a program is safe when we can assure that the formal definition of our specification is complete. Consequently, talking about verification coverage also means measuring the completeness of our specification. An approach like mutation coverage can realize this kind of coverage idea. It is defined as making changes to a system model and subsequently checking whether a given property set can spot those modifications. There are two categories of mutation techniques. Where semantic mutation describes a change in values of elements, like replacing variable values in a state with different ones, structural mutation focuses on modifying the structure of a model by replacing a state or a transition [21].

2.2.2 Completeness of Verification Procedure

Checking the completeness of properties is an attempt to gain information to improve the specification. On the other hand, it can be interesting also to process data regarding the verification procedure itself. Since model checking suffers from the state-explosion problem, there are scenarios where the verifier does not come to a conclusion. As a consequence, the verdict is neither true nor false; instead, it is unknown. As a verification engineer, it can be beneficial if more data regarding the result outcome exists. This is reasoned by the idea of instead continuing the verification on the point it failed instead of starting from the beginning [4]. Approaches like this already exist in the way of giving them information that one model checker produces as input to another model checker such that the verification problem is limited to the reduced state space. This means that the state space only consists of the uncovered part from the first model checker. In literature, experiments prove this approach's effectiveness by showing a significant improvement of the verification result and improvement of the performance [4].

Building upon this idea, there are some more abstract approaches to this problem by defining a measure that depicts the progress of the verification procedure. There are ideas based on abstract reachability trees to compute an under-approximation of such measure [8]. One realization is taking as input an assumption automaton and giving the verification coverage as output. The computation is done by encoding the negation of the conditions under which a statement is covered as a safety property. The approach of under-approx. is used to compensate performance penalties of the calculation [8].

2.3 Classification of this Thesis

When classifying this thesis alongside the broad spectrum of research fields of verification coverage measures, it belongs to the domain of software model checking, more precisely, to the category of checking the completeness of properties and verification procedure. The first category applies since we show in this thesis verification coverage approaches which indicate if some parts of the program were not considered during the analysis. Regarding the second category we try to improve the information output at the end of an unfinished analysis to help the verification engineer better handle unknown cases. In addition, a new approach that is less discussed in the literature regarding the development of the verification coverage during the analysis is proposed. This idea tries to help evaluate the whole verification procedure according to its effectiveness since it shows at which period the coverage increase was at its highest.

3 Background

Before we discuss the new proposed measures, we need to clarify first the theoretical background of this thesis. In addition, we give a small overview of the tools used to implement the ideas.

3.1 Abstract Measure Definition

In literature the term "coverage metric" is often used which does not confirm to the standard mathematical definition of metric. For that reason, we use instead the term "coverage measure", since the standard mathematical definition for measure is more appropriate in this context. Therefore we consider a measure as an abstraction of the length of an interval in $\mathbb{R}$ [16].

3.1.1 σ-Algebra

Let X be a set. A nonempty system A of subsets from X is called σ-algebra if $a \in A$ implies $X \setminus a \in A$ and $a_1, a_2, ..., a_n \in A$ implies $\bigcup_{i=1}^{\infty} a_i \in A$.

3.1.2 Measure Properties

Formally we define a measure as a function $\mu : A \to \overline{\mathbb{R}}_+$ where $\overline{\mathbb{R}}_+$ stands for the expanded set of positive real numbers which includes 0 and $+\infty$ and μ is defined on a σ-algebra A. In addition following properties must apply:

1. Non-negativity: $\forall a \in A : \mu(a) \geq 0$

2. Null empty set: $\mu(\emptyset) = 0$

3. σ-additivity: $a_1, a_2, ..., a_n \in A$ with $a_k \cap a_l = \emptyset$ and $k \neq l$
   implies: $\mu(\bigcup_{n=1}^{\infty} a_n \in A) = \sum_{n=1}^{\infty} \mu(a_n)$

3.1.3 Normalization

In addition to the mathematical definition, we specify an additional fourth property.

4. Normalization: $\forall a \in A : \mu(a) \leq 1$

This is most often achieved by taken the theoretically maximum possible measure value as divisor: $\mu_{normalized}(a) = \frac{\mu(a)}{\mu_{max}}$

Ultimately, this property is not necessarily required to define a measure. Nevertheless, we apply it for all proposed coverage measures to make them better comparable to each other.

3.1.4 Verification Coverage Measure

Verification coverage measure in this thesis means that we build upon the mathematical definition of measures and apply this to verification coverage. Therefore, mapping from verification-specific data to the set of real numbers is necessary. We use Function Domain (3.1) for this mapping where the input domain $CoverageData$ is verifier and analysis specific. We consider every data structure used during the verification as potential element of $CoverageData$, like for example a set of predicates or a set of code lines.

$$count : CoverageData \rightarrow \mathbb{R}_+ \tag{3.1}$$

The concrete semantic meaning for the abstract mapping function *count* generally depends on the measures. In our case, we use for the later proposed coverage measures as concrete definition for $count(C)$ the cardinality of C, where C is a set, consequently resulting into Function (3.2).

$$count(C) = |C| \tag{3.2}$$

3.1.5 Growing Coverage Data

We differentiate between so-called growing coverage data and non-growing coverage data. Growing coverage data are a collection of elements or a single scalar value where the collection size or, respectively, the value only increases during the verification analysis, whereas non-growing coverage data can vary on

Example C-code snippet

```
if (x > 0) {
  y = x;
} else {
  y = -x;
}
i = 1;
while(i < y) {
  i = 2 * i;
}
```

Corresponding CFA

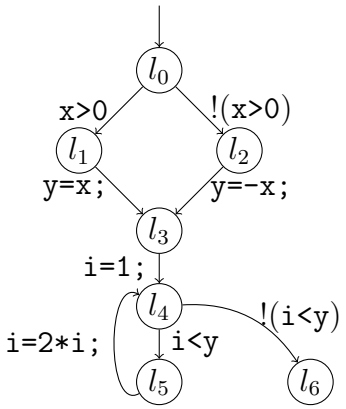


Figure 3.1: CFA for a given C-code snippet.

collection size or, in the case of a scalar, in its value during the verification analysis. When reconsidering the σ-additivity characteristic, we can deduce for the measure a monotonic growth from growing coverage data. This kind of coverage measure development fulfills the idea that the more time the verifier has, the more the coverage increases, or at least it staggers at some point. This behavior is preferable. Otherwise, when we would allow a decrease of coverage at some time during the analysis, that would be counter-intuitive. When we look at the analogy of test coverage, that would mean that the coverage would decrease if we run through a test suit and add further tests. Therefore we focus in the next chapter on growing coverage data.

3.2 Control Flow Automaton (CFA)

We take the basic definitions from the literature [3]. A control-flow automaton (CFA) is a three-tuple $P = (L, l_0, G)$ where L is the set of program locations with $l_0 \in L$ as the initial location and G represents all control-flow edges $G \subseteq L \times O \times L$. We call locations $l \in L$ also CFA nodes. The set of operations O generally has two types of operations: assumes and assignments. An example how a CFA is constructed from a C-code snippet is depicted in Fig. 3.1 where on the left side is the code and on the right the control-flow automaton. We will later consider CFA nodes as potential candidate in the calculation of some measures.

3.3 Configurable Program Analysis

In the following, we want to define important terms regarding the verification analysis which serves as our environment where we define our coverage measures. Since we use CPAchecker[1] [6] as a verification framework where we implement the upcoming approaches, we need to look closer at Configurable Program Analysis (CPA) [5].

3.3.1 CPA

A CPA is defined as a four-tuple $\mathbb{C} = (D, \rightsquigarrow, merge, stop)$ and operates on a CFA $P = (L, l_0, G)$. Its four components, the abstract domain D, the transfer relation $\rightsquigarrow$ operator, the *merge* operator and the *stop* operator are explained in the following sections.

3.3.2 Abstract Domain

An abstract domain D is defined as a three-tuple $D = (C, \mathcal{E}, \langle\cdot\rangle)$. The semi-lattice $\mathcal{E}$ is an algebra consisting of a set E and a partial order operation $\sqsubseteq$ satisfying associative, commutative and idempotent properties [9]. The set E represents during the analysis the abstract states of our program. The concretization function $\langle\cdot\rangle$ is defined as: $\langle\cdot\rangle : E \to 2^C$ and maps an abstract state to a set of concrete states C. Concrete states can potentially be reached by the program, whereas abstract states represent multiple concrete states.

3.3.3 Transfer Relation Operator

A transfer relation is used to traverse through abstract states. It computes all possible abstract successors of an abstract state and is defined as $\rightsquigarrow \subseteq E \times G \times E$ where E represents the set of abstract states.

3.3.4 Merge Operator

The merge operator decides when and how abstract states are combined. Formally it is defined as $merge : E \times E \to E$. To fulfill the correctness criterion, the second parameter must be subsumed: $\forall e, e' \in E : e' \sqsubseteq merge(e, e')$. Otherwise the new abstract state is an over-approximation of both input states.

[1] `https://cpachecker.sosy-lab.org`

Algorithm 1 CPA algorithm taken from [5]

Input: $CPA = (D, \rightsquigarrow, merge, stop), e_0 \in E, CFA = (L, l_0, G)$
Output: **Set of reachable abstract states**
Variables: $reached, waitlist$

```
1: waitlist := {e_0};
2: reached := {e_0};
3: while waitlist ≠ ∅ do
4:     pop e from waitlist
5:     for all e' with e ⇝ e' do
6:         for all e'' ∈ reached do
7:             e_new := merge(e', e'')    ▷ Combine with existing abstract state.
8:             if e_new ≠ e'' then
9:                 waitlist := (waitlist \ {e''}) ∪ {e_new};
10:                reached := (reached \ {e''}) ∪ {e_new};
11:            end if
12:        end for
13:        if ¬stop(e', reached) then
14:            waitlist := waitlist ∪ {e_new};
15:            reached := reached ∪ {e_new};
16:        end if
17:    end for
18: end while
19: return reached
```

3.3.5 Stop Operator

The stop operator is formally defined as $stop : E \times 2^E \rightarrow \{true, false\}$ and decides when to stop exploration, meaning if newly computed abstract states should be added to the waitlist or not.

3.3.6 CPA Algorithm

The CPA algorithm depicted in Algorithm 1 starts by initializing the waitlist and reached set with the initial element e_0. We then compute the abstract successors via the transfer relation $\rightsquigarrow$ for the next element from the waitlist. Afterward, the merge is performed for every successor with every existing element reached. When we have a new element e_{new} generated by the merge operator, we replace the existing element [5].

3.3.7 Abstract Reachability Graph (ARG)

An abstract reachability graph is a representation of an abstract state space. It is used as proof of the performed verification, and in case of property violations, it is suitable for extracting error paths. All together we can now define formally an ARG for a CFA $P = (L, l_0, G)$ and a CPA $\mathbb{C} = (D, \rightsquigarrow, merge, stop)$ as: $ARG = (N, root, \mathbb{G})$, where N is a set of abstract states which were reached during the CPA algorithm, $root$ is the initial abstract state and $\mathbb{G}$ is defined as $\mathbb{G} \subseteq N \times G \times N$. We call $g \in \mathbb{G}$ an ARG edge and $n \in N$ an ARG node. In addition the following structural properties must apply: $root \in N$ and $N \subseteq E$.

3.3.8 Predicate Abstraction

Predicate abstraction [12] is a reachability analysis traversing the CFA. During this process, an ARG is generated. Predicates over program variables from a given precision set represent the abstract states. We can create successor abstract states from a given abstract state with an operation by computing a boolean combination of predicates from the precision.

3.3.8.1 CEGAR

Counterexample-guided abstraction refinement (CEGAR) [11] is an approach to increase the performance of the verification process. This means that we have for our program P an initial precision π_0 which approximates the reached states. When we reach an error state, we can construct a counterexample from the ARG and check if it is feasible. If true, we have found a property violation, and therefore, we can end the analysis; if it is spurious, we do an abstraction refinement, which leads to new predicates. With this updated, refined precision π, we repeat the analysis as long as we do not find any new counterexamples.

3.3.8.2 Lazy Abstraction

Lazy abstraction [14] improves the performance by not completely deleting the previous computed ARG after the refinement step of the CEGAR algorithm. It recomputes only those parts of the ARG where the new predicates are necessary. In addition, new generated predicates will not be used globally for all abstraction computations. In detail, this means that lazy abstraction narrows the scope on relevant parts of the ARG or CFA regarding the usage of predicates.

3.4 CPAchecker

We use the verification framework CPAchecker to realize all of our later approaches. It is based on the already discussed CPA algorithm and has a modular design to add further CPAs. This allows configuring the verification analysis to specific variants like predicate analysis with CEGAR or value analysis. Both have their advantages and disadvantages. We will later propose some coverage measures for predicate analysis. Regarding our analysis-independent approaches we will also use value analysis for comparison. The analysis type can be configured in the command line settings of CPAchecker.

3.4.1 Predicate CPA

When using CPAchecker, we specify the use of the Predicate CPA module in the command line with the following option: `-predicateAnalysis`. If we do so, the analysis uses predicate abstraction to compute all abstract states. Consequently, we use a set of predicates to decide which parts of the program are reachable during the analysis. Per default, the option for using CEGAR is activated to increase the performance of the verification process. It is important to note that if we talk later about the predicate set we also consider lazy abstraction.

3.4.2 HTML Report

When using CPAchecker to analyze a simple C-program we can specify the option to generate an HTML file typically called `Report.html` or in case of a property violation outcome `Counterexample.html` to view details regarding the verification result. The CFA tab (cf. Fig. 3.2) displays a visual representation of the control flow automaton used during the analysis. The tab-specific toolbar below allows further customization regarding the visuals of the displayed CFA. There are also other tabs within the report, like for example the Source tab (cf. Fig. 3.3). Here we can select any of the input source files and display its content. As part of the realization of the later proposed visualization approaches we will integrate additional visualization capabilities to both mentioned tabs.

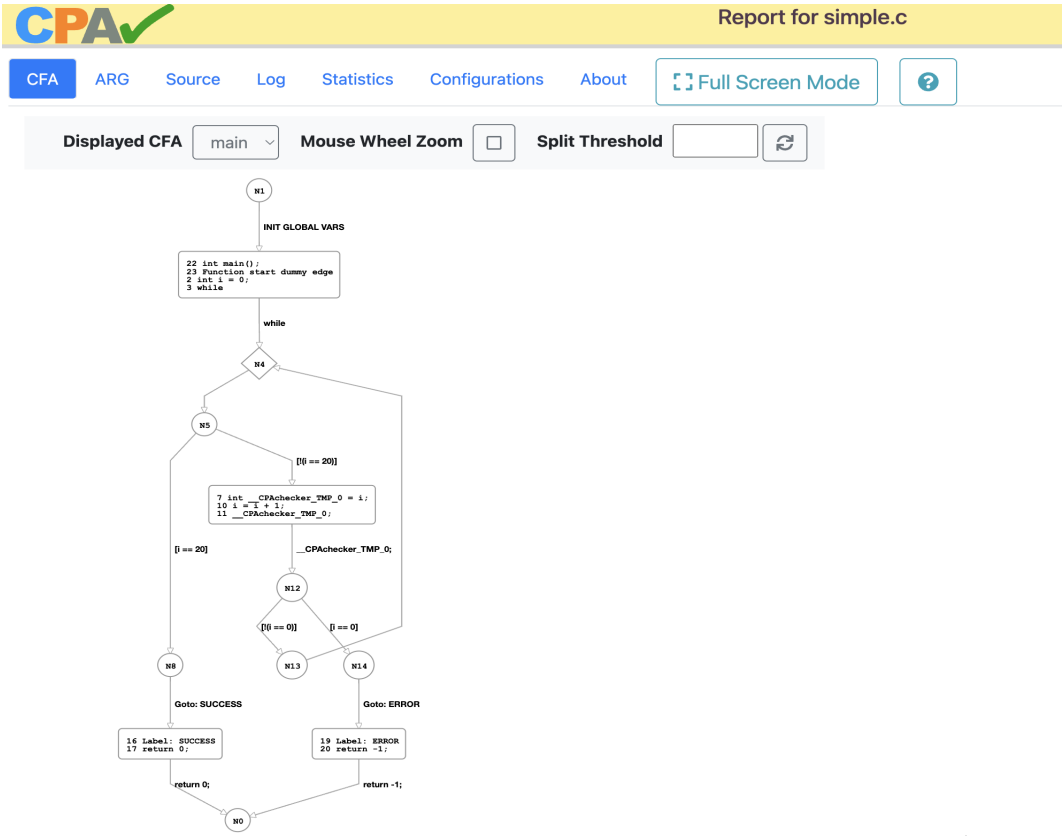


Figure 3.2: CFA tab in Report.html.

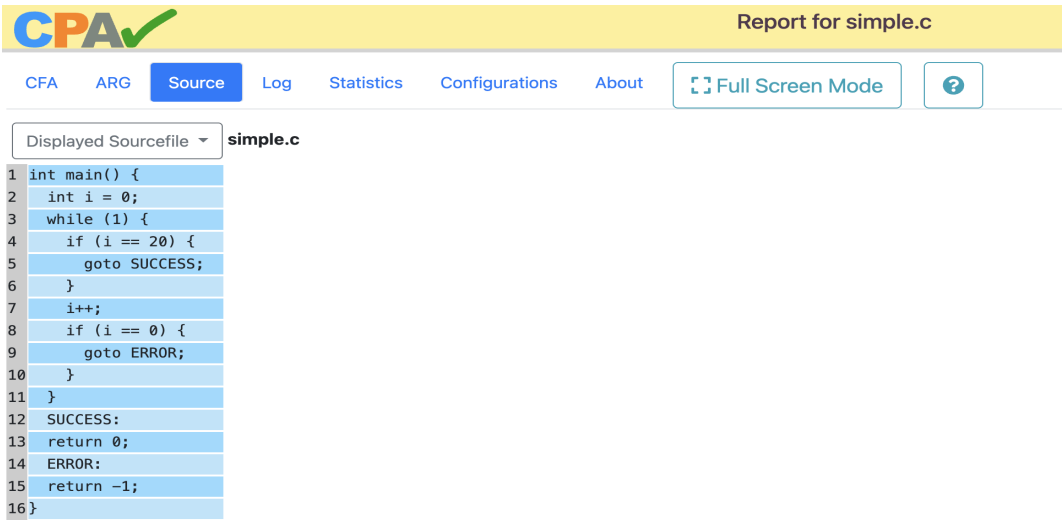


Figure 3.3: Source tab in Report.html.

4

Verification-Coverage Measures

Based on the so far discussed theoretical background, we want now to propose new approaches for verification coverage. First, we look at the different abstract categories of verification coverage. Second, we discuss concrete approaches based on the mentioned categories, resulting in new measure definitions. Lastly, we build upon the idea of measures and define the concept of *time-dependent coverage* (TDC) to allow the tracking of the progress of coverage over time. The goal of this chapter is to understand more behind the theory and realization of verification coverage measures allowing the verification engineer to gain more information from the analysis.

4.1 Categorization of Verification Coverage

4.1.1 Application Purposes

When we talk about verification coverage software model checking, we should first differentiate between possible application fields. As already mentioned in the Literature Overview chapter, we consider two categories: *completeness of properties* and *completeness of verification procedure*. The first one has the goal of determining the verification coverage of the program at the end of a completed analysis. The resulting number should help the verification engineer to interpret how complete their specification was or, in other words, how much of a specific program property was considered by the verifier during the anal-

ysis. This could help to find issues in the used specification because, e.g., we could have a bad specification design and, therefore, some code fragments are not considered. Consequently, the verifier is not checking the whole program and we would miss some potential bugs. An untypical low coverage value could alarm the verification engineer to check the used specification if it is correctly defined for the given task.

Completeness of the verification procedure, on the other hand, has the goal to determine how far the analysis was when we had an unknown case. Finding a measure for accomplishing this goal and only relying on that resulting number is difficult to interpret since verification tools typically do not have a common interface delivering data to deduce the progress. Therefore we work with various visualizations to give a better overview of the unfinished analysis. This can be accomplished in many different ways, which we will discuss later.

4.1.2 Output Domains

For defining our output domain based on measurement theory, we need to consider the levels of measurements [23]. We list here some of the most commonly used ones and discuss them.

4.1.2.1 Nominal

When using nominal output data, we have distinguishable categories, for example when measuring colors. We could have data like red, green or blue but we have no natural rank order. As a consequence, we cannot compare single data elements to each other. This is not suitable for verification coverage since we want the possibility to detect comparable changes in coverage for different scenarios, allowing us to interpret the result in a way such that we can conclude an improvement or worsening.

4.1.2.2 Ordinal

When adding a natural rank to nominal data, we have ordinal data, meaning that we can compare different output elements. An example output domain for our use case would be a set consisting of the outcomes: “no coverage”, “partial coverage”, and “full coverage”, where the first mentioned case has the lowest rank and the last one the highest rank. This level of measurement can be useful as a verification coverage output domain but is limited when we want to determine the degree of difference between two values.

4.1.2.3 Interval

Interval data builds upon ordinal data and adds quantification. Consequently, we can calculate the difference between two values. This scale still has the problem that we could not say that coverage value A is double the value of coverage B. This is due to the freedom of having an arbitrary origin.

4.1.2.4 Ratio

In contrast to interval data, ratio scales have a natural non-arbitrarily chosen origin. Therefore any linear transformation can be applied. This level of measurement fits best to our measure definition Function (3.1) since it does not contradict any of our mentioned measure properties. When using the normalization property, we output numbers between 0 and 1, indicating how much of the program was covered by the verifier. The number 0 is the origin and stands for "the program was not covered", and 1 for "the program was fully covered". Values in-between indicate a partial coverage where higher numbers are interpreted as higher coverage, meaning that a coverage value of 0.6 indicates triple the coverage of 0.2.

4.1.3 Input Domains

We distinguish between two types of possible input data. The first category is all data generated by the verifier but independently of the verification coverage measure approach, like a CFA or an ARG. We call this category *result-specific data*. Working with this kind of data has the advantage of allowing post-processed verification coverage calculation since the data is already existing and fully available at the end of the analysis. In the other case, we have input data for the measurement calculation, which must first be generated during the analysis. Those depend on the verification algorithm details. We call this category *coverage-specific data*. To illustrate this categorization, we look at the core algorithm of CPAchecker. For example, the transfer relation operator can be interesting to consider. This is reasoned by the idea that we prefer growing coverage data. Finding those data needs operators or algorithm fragments that generate continuously states. When looking closer at the CPA algorithm depicted in Algorithm 1 we see two phases where new data results and gets potentially saved later in the waitlist or reached set. One at line 5, where we get e' with the transfer relation operator $\rightsquigarrow$; the other on line 7, where we get e_{new} with the merge operator. An abstract approach for deducing growing coverage data based on the mentioned data can be processing the occurrences with other additional restrictions of the data. In this case, we would make use of the $count(C)$ function from the background chapter. Candidates as

input for the $count(C)$ function could be a set of abstract states or a set of CFA edges which we have gathered during the analysis with the help of the transfer relation operator. Therefore $count(C)$ gives us as output value some real number representing, for example, the amount of different code lines, CFA locations, abstract states, assume edges, or any other coverage data we have specified in its implementation. Comparing both categories of possible input data leads to the following conclusion. While the first type is, in most cases, more accessible to implement, the second one has the potential to provide a broader domain of data to use for calculation.

4.1.4 Analysis Dependencies

We differentiate our approaches between *analysis-independent* and *analysis-specific* verification coverage. While analysis-independent verification coverage can be applied to any model checking verification analysis which works with data structures similar to CFAs and ARGs, we also narrow our scope on verification coverage measures for specific analysis. This allows us to use further specific details to improve the coverage calculation. For the analysis-specific category, we consider in this thesis predicate analysis with CEGAR.

4.1.4.1 Verification Coverage with Predicate Analysis

During predicate analysis, we use predicates over program variables from the given precision π to determine the feasibility of paths. This precision set continuously grows throughout the verification analysis. When using predicate analysis in combination with CEGAR, we have in the beginning an initial precision, typically an empty set of predicates, which we refine whenever we reach an error location. A refinement normally leads to an increased predicate set size. Therefore it fulfills the growing coverage data definition. Both mentioned sets are not only technically qualified to be used as the basis for the measurement definition, they also semantically fulfill the idea of giving an approximation about the coverage of the program. This is due to the fact that adding new predicates is based on exploring new states or, in the case of CEGAR, finding a counter-example path. Consequently, continuously adding new predicates indicates progress in finding new interesting abstract states. Therefore we can build up a connection between new predicates and coverage. Taking this concept one step further, we could also consider the specific content of the predicates, which is mostly built from variables based on the program code. Consequently, we could work with the logical formulas or the occurring variables.

4.2 Approaches for Verification Coverage

Combining the theoretical knowledge about verifiers and measures with the concepts of verification coverage, we can now propose new approaches. We also implemented all the following verification coverage measures in CPAchecker to test them in practice. The way how to implement in general measures is described in the Implementation chapter. Later in the Evaluation chapter, we also compare all of these realizations.

4.2.1 Completeness of Properties

4.2.1.1 Visited-Lines Coverage Measure

For completeness, we first start with the most basic idea, which already exists in CPAchecker. Nevertheless, we want to explain this coverage measure here, since some others are building up based on this approach. *Visited-lines* coverage can be implemented using result-specific data as well as coverage-specific data, since it only depends on the ARG. The latter approach has the benefit that we could calculate a temporal coverage value, since coverage-specific data is available during the analysis. Tracking temporal coverage values has the advantage that we can generate a *time-dependent coverage* (TDC) data series, which we will discuss in the next section. Furthermore, *visited-lines* coverage is an analysis-independent coverage approach. Lines in our context means code lines from a given source file. We differentiate between lines which contains statements and those which are blank or contain only syntactical elements like a closing bracket. The first one we call *statement lines* the latter *blank lines*. We consider a line visited if we have an ARG edge containing a statement line. The set of visited lines represents our input set C for the $count(C)$ function (cf. Function (3.1)). As a normalization divisor, we take the total number of statement lines and call it *TotalLines*. Taking all together we use Equation (4.1) for calculating the coverage, where *VisitedLines* is a set of code lines considered as visited.

$$Coverage(VisitedLines) = \frac{count(VisitedLines)}{TotalLines} \in [0, 1] \tag{4.1}$$

As the output domain, we use a ratio scale from 0 to 1. This is an appropriate measurement level since we have a natural origin (no visited lines), and all values represent a quantification. Double the coverage means we have visited double the lines. *Visited-lines* coverage is suitable for simple analysis where we do not use precision refinement techniques or restarts. Otherwise, for analyses which use for example CEGAR, we could have the issue of over-approximation of the *VisitedLines* set. Therefore our coverage number is higher than the true

value we are targeting to determine. On the other hand, we still can conclude that the real coverage is at most that calculated number or less. Consequently, our computed visited coverage represents, in this case, an upper limit.

Visited-lines coverage can be applied for the task completeness of properties. Suppose we have a method or code block for our analyzed program that does not get considered by the verifier, but we actually want it to be checked, in that case, a coverage value less than 1 should result since not all lines were visited by the verifier. Especially combined with a visualization approach that marks all visited lines of the source code, we should be able to quickly detect parts of the code that were not covered.

It can also be used for completeness of verification procedure but has the issue that, as already mentioned, it over-approximates fast, and therefore we would have in most unknown cases a high coverage, which would not tell us how far the verification procedure actually was. This approach should deliver coverage values that can be interpreted in a helpful way in the context of completeness of verification procedure for simple programs without loops and analysis types that do not over-approximate the *VisitedLines* set. Since these cases rarely occur, we classify *visited-lines* coverage to the category of completeness of properties.

4.2.1.2 Visited-Variables Coverage Measure

Visited-variables coverage measure works similarly like the *visited-lines* coverage measure. It is also an analysis-independent approach and considers visited lines. Instead of storing them in a set, we check if a new variable is declared within the line. Suppose we have a new declaration, then we save the variable name, including its function scope, to the *VisitedVariables* set. When the analysis is finished, we count all occurring variables within the given source code. We call this number *TotalVariables* and use it as a normalization divisor.

$$Coverage(VisitedVariables) = \frac{count(VisitedVariables)}{TotalVariables} \in [0, 1] \tag{4.2}$$

We then use Equation (4.2) for calculating the coverage. For this measure the same verification coverage categorization properties apply as for visited-lines coverage. A potential advantage of this approach is that the resulting coverage value is not influenced heavily by many not relevant source code lines. Not relevant in this context means that when we remove a code line, there would be no significant change in the program control flow. When we already have, for example, a C-program with print statements, and we want to add additional print statements for logging existing variables or just displaying at which program location we are, then the *visited-lines* coverage can change since we have potential new lines to cover, whereas *visited-variable* coverage

would stay stable since we did not declare new variables. We prefer for this scenario a more stable behavior since no relevant change in a program should not indicate a change in coverage. Otherwise, we could add so many print statements that the *visited-lines* coverage could approximate 1. Consequently, the coverage would hardly change if we add relevant lines.

Nevertheless, *visited-variables* coverage can have the same issue when we would add many unused variables. We mean with *unused*, variables that are declared but never used in the program. Therefore code lines that declare unused variables are also no relevant code lines. Our following approach is also based on variables and solves the issue of unused variables.

4.2.1.3 Predicate-Abstraction-Variables Coverage Measure

Another measure approach that belongs to completeness of properties category is *predicate-abstraction variables* coverage. It is an analysis-specific approach and therefore depends on predicate analysis. We use the coverage-specific data set *PredicateAbstractionVariables* as the input domain. This set can be generated during the analysis. The idea here is to gather all variables which occur in the abstraction formula of the current predicate abstract state, which is visited by the transfer relation. In addition, we count at the end of the analysis all program variables of the source code which we then use as normalization divisor (*TotalVariables*) for the coverage calculation (cf. Equation (4.3)). We could also look at the end of the analysis at the ARG states, but we prefer the first approach since it facilitates the time-dependent coverage (TDC) data series generation which we will discuss in the next section.

$$Coverage(PredicateAbstractionVariables) = \frac{count(PredicateAbstractionVariables)}{TotalVariables} \tag{4.3}$$

When we have a finished analysis, we expect a relatively high coverage value since we assume that most of the variables should be considered during the analysis. If not, we have an indication that some program parts were not processed by the verifier since some variables were missing in the abstraction formula. We then would check if our used specification was suitable for our verification task. Combined with a visualization approach which we discuss later, we could depict all missing variables and where they are located. Consequently, we could check the plausibility of the result, meaning if it was legitimate that the verifier did not process some variables. The output domain is of type ratio data. This is suitable since we have a natural origin (no visited variables), and all values represent a quantification.

4.2.2 Completeness of Verification Procedure

4.2.2.1 Predicates-Considered-Locations Coverage Measure

As we already mentioned for *visited-lines* coverage, the problem for using it in case of completeness of verification procedure is that it over-approximates too fast. Therefore our motivation is to find an approach that reduces the over-approximation of coverage, leading to more meaningful results in unknown cases. We consider a setting where we use predicate analysis with CEGAR. Consequently, we have a wider choice regarding the selection of a suitable coverage data as a basis for the $count(C)$ function compared to an analysis-independent approach. More precisely, we now take the predicate set into account. After each refinement during the analysis, we potentially add new predicates into this set. Since we do not remove any of these predicates, it fulfills the growing coverage data criteria.

As the input domain for the coverage calculation, we choose a coverage-specific data called *PredicateConsideredLocation*. As the name of the set already implies, we now do not consider code lines, instead we work with CFA locations. The advantage of this consideration is that we can later deduce helpful visualizations of the CFA from this coverage-specific data set. This also has the side-effect of reducing the implementation complexity since we do not have to deduce corresponding code lines from the CFA edges. Similar to *visited-lines* coverage, we use visited CFA edges but instead considering the referencing code lines, we look at its successor CFA node and add them to the *PredicateConsideredLocation* set if the *predicate-considered requirement* is fulfilled. This restriction is the additional component of the algorithm that is missing in *visited-lines* coverage and tries to compensate for over-approximation regarding the computation of the verification coverage measure to minimize this disadvantage.

The *predicate-considered requirement* is only applied on CFA edges holding an assume-statement. The idea is that we want to sort out over-approximated CFA path choices. If we have an assumption consisting of variables that are not contained in any of the formulas of the set of predicates from the given precision, we have a high chance of over-approximating since we would visit locations along the CFA path without safely knowing if they are reachable. Therefore we only add CFA locations within the *PredicateConsideredLocation* set if either they have no incoming CFA assume edge or if all assume variable names are contained within the set of all predicate variable names we construct from the mentioned predicate set. In addition, we require that every existing direct predecessor CFA location is also element of the *PredicateConsideredLocation* set. Ideally, we then should have, as a result, a subset of the *VisitedLocations* set.

Algorithm 2 Predicate-Considered

Input: $predicateVariables$, $cfaEdge$
Output: $\{true, false\}$

```
1: if cfaEdge has type AssumeEdge then
2:     assumeVariables := convertAssumeToVariables(cfaEdge);
3:     if assumeVariables ⊂ predicateVariables then
4:         return true;
5:     end if
6:     return false;
7: end if
8: return true;
```

As normalization divisor we take the total number of CFA locations and call it *TotalLocations*. The calculation of the measure is defined in Equation (4.4).

$$Coverage(PredicateConsideredLocations) = \frac{count(PredicateConsideredLocations)}{TotalLocations} \tag{4.4}$$

This approach is a fast approximation of ruling out potentially unfeasible paths, which nevertheless leads to over-approximation regarding the coverage value but in contrast to *visited-lines* coverage at a decreased factor. If we would further improve this approach regarding the approximation, we would need to calculate if the current visited path is feasible. This is not reasonable since it would decrease the performance significantly. When looking at Algorithm 2 we see the that the performance of the algorithm depends on the subset check in line 4. Checking for a subset takes on average linear time when using a hashtable for the implementation of the sets.

The output domain of the coverage is between 0 and 1 on a ratio scale since our basis of calculation has a natural origin (no locations considered) and supports quantification.

4.2.2.2 Predicates-Relevant-Variables Coverage Measure

The *predicates-relevant-variables* coverage measure is an enhancement of the *predicates-considered* coverage measure. It is based on the same premises. This means it is also based on predicate analysis and uses coverage-specific input data for the coverage calculation, the *PredicateConsideredLocations* set.

In contrast to the measure before, the idea behind this approach is to maintain a set of relevant variables called *RelevantVariableNames*. *Relevant variable* in this context means that it is helpful for us regarding the decision if we over-approximate a program path. The set is built by retrieving

all predicates from the current program location and then applying a filter to this set. The filter can be realized as a heuristic that removes potentially not relevant variables. Our approach uses a frequency heuristic, where relatively rarely occurring variables are excluded. Suppose we have a CFA edge holding an assume-statement, and we have, like in *predicate-considered* location coverage, also a predicate formula containing all variables of the assume-statement. We then still could over-approximate since the formulas holding those variables could not be strong enough. Therefore the idea of our heuristic is that the more predicate formulas we have for one variable, the higher the chance that we do not over-approximate since we then would have more information. Therefore we consider only variables as relevant if they occur relatively often within our set of predicate formulas.

We define the frequency threshold parameter, which determines if we consider a variable as relevant, as a number between 0 and 1, indicating the percentage of how much of all variables we want to keep depending on their relative frequency. Finding an appropriate threshold to apply the variable exclusion is open and needs to be determined, e.g., by experiments. When removing the filter or setting the parameter value to 1, this measure behaves like the *predicates-considered* coverage measure.

4.3 Time-Dependent Coverage (TDC)

Until now, we had considered verification coverage as a resulting number when the analysis was over. There are cases, especially in the context of completeness of verification procedure, where it could be interesting to know how the actual development of this coverage was during the analysis.

4.3.1 Basic Idea

Knowing at which interval the verification coverage had a high growth or was staggering allows to understand the whole analysis process better and makes it easier to compare coverage measures. The first question is which dimension is suitable for making the coverage data dependent. Basic ideas are verifier steps or time. Verifier steps, in this context, is an abstract term. We could define it, for example, as one step representing one call of the transfer relation operator. In our approach, we consider the time dimension, in detail, the wall clock time. Therefore our idea can be implemented for any verifier since measuring the wall clock time should always be possible. From now on, we are talking about *time-dependent coverage* (TDC).

4.3.2 Implementation

Implementing TDC needs a data collection that consists of tuples. These tuples contain a timestamp and the corresponding verification coverage value at that time. This implies that it is not sufficient to calculate the coverage at the end of the analysis, instead we need to do it whenever we add a new tuple. Adding a new tuple should be considered at that point where we potentially can have a change in calculating the verification coverage. For our proposed measures it is sufficient to add new tuples after each transfer relation operation.

4.3.3 Requirements

The mentioned implementation approach, therefore, requires measures that are based on coverage-specific data. Measures based on result-specific data are for TDCs not suitable since we cannot deduce at the end of the analysis from the given data at which timestamp they were added. We could potentially construct it in chronological order, but without time data we do not know the time span between two data elements. Therefore, it is required to collect the time data during the analysis, combined with the coverage-specific data.

4.3.4 Visualization

Working with a visualization facilitates the comparison of different measures and detection of structures for the human eye. When the analysis is over, we can use our generated TDC data series to render a graph, which we call a *time-dependent coverage graph* (TDCG). The horizontal axis represents the wall-clock time. It starts from zero and ends at the time representing the duration of the analysis. On the other hand, the vertical axis stands for the verification coverage, ranging from 0 to 1. Since all of our proposed coverage measures are based on growing coverage data, we expect to see a monotonic graph development beginning from zero and ending at the data point representing the tuple, consisting of the analysis duration and the final coverage value. We depict later in the implementation and evaluation chapter some example TDCGs.

5 Verification-Coverage Visualization

So far, we have described verification coverage as a single number or TDC. Often it is preferable for the verification engineer to work with a visualization of coverage. This helps better understand more complex cases in a single view. Therefore we propose visualization approaches for different components relevant to the analysis itself. In addition, we discuss already existing techniques for comparing purposes.

5.1 Source Code Lines Visualization

One category of visualization approaches is source code coloring. This technique is already known in software testing, where we run test coverage tools that show us which line of code was executed by certain tests and which were not. In verification coverage, this idea is similarly realized. The difference is that it is not apparent what kind of criteria to use when deciding to color a line. Therefore we start with an algorithm for source code coloring, which is already known and implemented in some verification coverage frameworks. We try to optimize the workflow and discuss the idea of working with this visualization technique.

5.1.1 Visited-Lines Heat-Map Coloring

The approach of coloring source code lines depending on visits is already a known concept used for test coverage tools and verification frameworks like CPAchecker. Visit in our context means that the verifier has considered a code line from a CFA edge during the transfer relation operation. Counting the total number of visits per code line allows generating a heat map by coloring the relative frequency of each line. In CPAchecker, there is the option of generating a `coverage.info` file after the analysis is done. This file holds the data of visits per line and complies with the LCOV coverage data file structure specification. For better readability, we can use the command `genhtml`, which is capable of generating an HTML view from our `coverage.info` file.

5.1.1.1 Improvement

As an alternative, we have implemented a similar visualization approach based on the same input data but integrated into the `Source` tab of the `Report.html`. This HTML file can be generated by CPAchecker after the analysis is done and does not need further post-processing. The goal here is to facilitate the use of coverage visualization for the verification engineer who inspects the final report. Therefore, it is no longer necessary to consider other tools for generating a proper visualization. If the coloring is not needed, a selector for enabling the default colors exists. *Visited-lines heat-map coloring* is automatically integrated if the `CoverageCPA` is specified in the CPAchecker configuration and the option `-setprop 'shouldCollectCoverageDuringAnalysis = true'` is set within the run command. An example of how the source code gets colored when enabled is depicted in Fig. 5.1. The darker the green, the higher the relative visit frequency. On the other hand, the red color means that the verifier did not visit any CFA edges containing the red colored line but it would have been possible, since we have an edge within the CFA which contains that line. A white background color is interpreted as "the code line was not considered by the verifier". This typically happens with lines that only have some syntax meaning but no semantic meanings, like lines only consisting of the symbol "}" in the case of a C-program. We call this kind of lines *blank lines*.

5.1.1.2 Discussion

Visited-lines heat map coloring uses the concepts of the *visited-lines* coverage measure and instead of calculating a single number, it visualizes all covered lines. This has the advantage of allowing the verification engineer to detect parts of the programs which were not visited by the verifier. This information can help check the plausibility of the result at the end of a finished analysis.

5.2 CFA Visualization

In contrast to source code coloring, visualizing the locations of a CFA better facilitates gaining information regarding the internal processing of the verifier. Two reasons cause this. First, we can better follow program execution paths. Looking closer at the CFA definition, this circumstance is evident since it is designed to show syntactically correct program execution paths. Therefore it is for the verification engineer more convenient to look at a CFA with verification coverage visualization to check which paths lead to program termination and are also covered by some criteria. Second, we have a better view of verifier internal automaton states. Therefore considering using a colored CFA can be beneficial for verification framework developers to debug issues better since they have more detailed internal verifier information. Nevertheless, using the source code visualization can be preferable if we solely want to focus on the code lines without considering internal verification details.

To illustrate the differences in visualizing the source code versus the CFA for the same code, we depict an example comparison in Fig. 5.1. We consider any green-colored line or node covered and every red-colored one not covered by the verifier. When looking at the CFA, we see two incoming edges leading to program termination at location N0. We can briefly detect that paths that end with the statement return -1 are not covered by the verifier, whereas all other ones ending with return 0 are covered. The verification engineer is now capable of spotting the beginning of the uncovered path fragment at N14 and the end at N20. Therefore we can conclude that this kind of analysis (in our example: value analysis) does not consider taking the branch `i==20` from N12. We can also see how verifier-specific statements like `_CPAchecker_TMP_0` which are contained within CFA edges between two covered locations are indirectly considered in the coverage visualization. In the following, we propose different new criteria for considering a CFA location covered by the verifier.

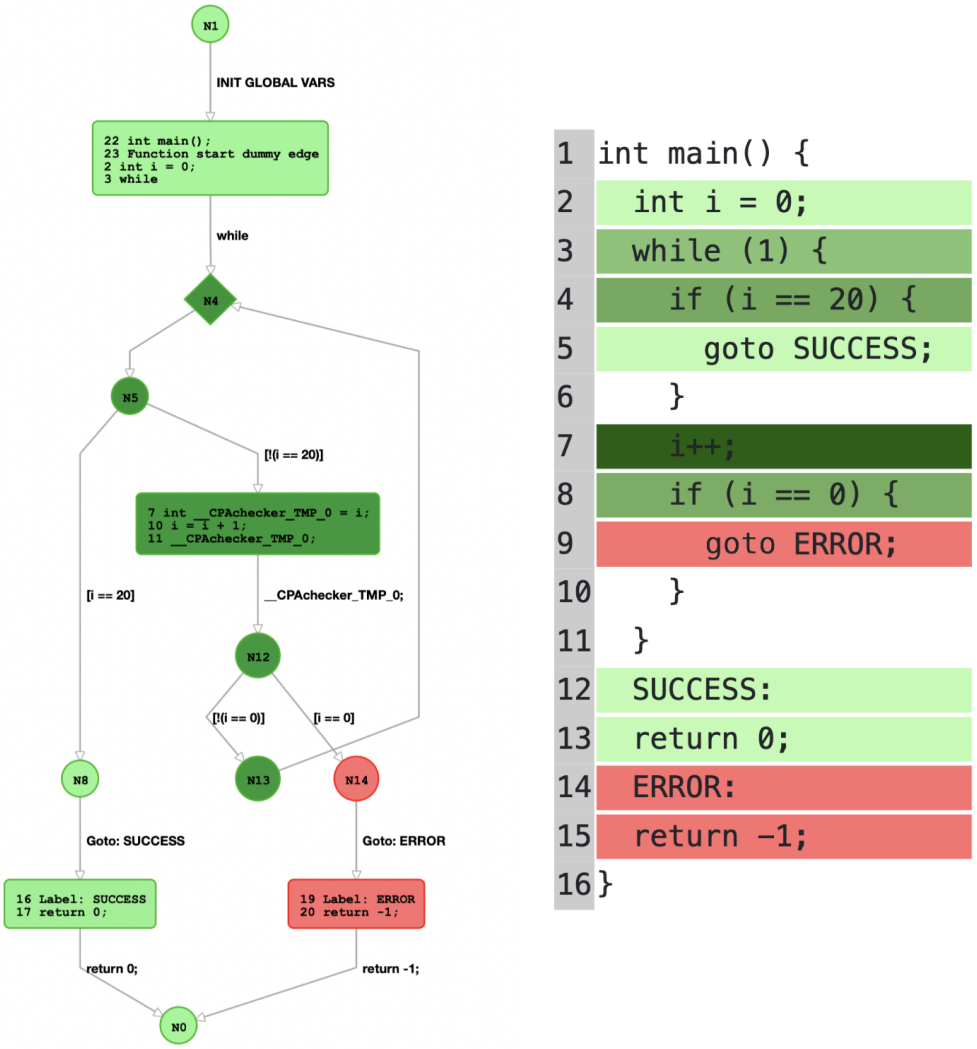


Figure 5.1: **Left:** CFA for the right code snippet; **Right:** Example C-code. Both visualizations were generated by CPAchecker with default value analysis.

5.2.1 Visited-Locations Heat-Map Coloring

5.2.1.1 Idea

The first and straightforward approach to considering a CFA location covered is when its incoming CFA edge was considered by the transfer relation operator during the analysis. If this happens, we call this CFA location visited. Like *visited-lines heat-map coloring*, we assume the verifier may need to visit the same CFA locations multiple times. Therefore we can raise the question of how often are the CFA location visited by the verifier during the analysis. Visiting a new location depends on the transfer relation operator determining the next abstract successors. Since the total number of visits for a specific location can heavily vary between different programs, we prefer to calculate the relative number of visits. This is done by finding the maximum number of visits for a CFA location and then normalizing all other total visit numbers per location. From this relative number which is within the range of 0 to 1, we then can color each location, where 0 means no coloring and 1 fully colored. For all locations with values in between this range, we apply a color gradient from no color to full color. When looking again at Fig. 5.1, we have used dark-green for locations that were relatively visited often and light-green for rarely visited locations. This kind of visualization allows us to briefly identify a dark-green path fragment, starting from CFA location N4 and then having the following path: $N4 \rightarrow N5 \rightarrow N7 \rightarrow N10 \rightarrow N11 \rightarrow N12 \rightarrow N13 \rightarrow N4$. This path indicates a loop since it starts and ends with the same location. Since this part of the CFA has locations colored in a darker tone than the others, we can conclude that the verifier indeed unrolled the complete loop during its analysis. This behavior is, in this case, typical for value analysis. When using predicate analysis with CEGAR, the resulting colored CFA would not have a darker green tone for the mentioned loop fragment. The reason for that is the initial low predicate precision which leads to an over-approximation. Therefore, all syntactically correct program paths are considered initially, leading to similar visit rates for each CFA location.

5.2.1.2 Discussion

Visited-locations heat-map coloring can be easily realized for most verifiers, which use a CFA and a transfer relation within their analysis. This approach benefits in detecting CFA locations that were visited often, for example, due to a loop. The developer of a verification framework could use this kind of visualization to compare potential optimizations on an analysis, whether some paths will be visited more often or not. On the other hand, we do not know if some of the visited locations were, in the end, reachable. This is due to the

fact that we could use an approach with over-approximation so that we visit some paths within the CFA which are not feasible. Consequently, it could happen, depending on the analysis, that we color most of the locations within the CFA. Therefore our next approach separates between visited and reached locations.

5.2.2 Considered-Locations Heat-Map Coloring

5.2.2.1 Idea

This approach is building up on *visited-locations heat-map coloring*. We now consider two different sets of CFA locations. The first is the set of visited CFA locations called *VisitedLocations* which results from the previous *visited-locations heat-map coloring* approach. The second one, called *ReachedLocations*, is filled with all locations which are reached. We construct this set by using the resulting ARG at the end of the analysis. We deduce all corresponding CFA locations from the ARG states and add them to the reached locations set. As the next step, we build a new set called *ConsideredLocations* which consists of the CFA locations which result when we take the *VisitedLocations* set and remove all elements which are also contained in *ReachedLocations* (cf. Equation (5.1)).

$$ConsideredLocations = VisitedLocations \setminus ReachedLocations \tag{5.1}$$

Now we apply the same visualization rules like *visited-locations heat-map coloring*, but with the exception that all CFA locations contained in the *ConsideredLocations* set are marked in another color, i.e., red. To illustrate this approach, we can look at Fig. 5.2. Here we use CPAchecker to construct a CFA from a bubble sort C-program (cf. Appendix A.1) and then apply our coverage visualization. In this example, we use predicate analysis with CEGAR. Note that this kind of analysis adds and removes states from the ARG. When the analysis has started, we manually terminate the verification process after a few seconds to have an incomplete analysis with the result "unknown". The dark green regions (i.e., N5-N10) within the CFA show that these locations are visited relatively often during the analysis. This is due to a loop that repeats for 100000 iterations. Apart from the reached green CFA locations, we also have a branch beginning from N27 and ending at N0, which is colored red. This part of the CFA was visited by the verifier but not reached during the analysis, meaning that there is no abstract state within the resulting ARG which has a corresponding CFA location. Consequently, we can deduce from this verification run that the verifier was processing locations somewhere in the red region before we interrupted the analysis. In addition, we also can

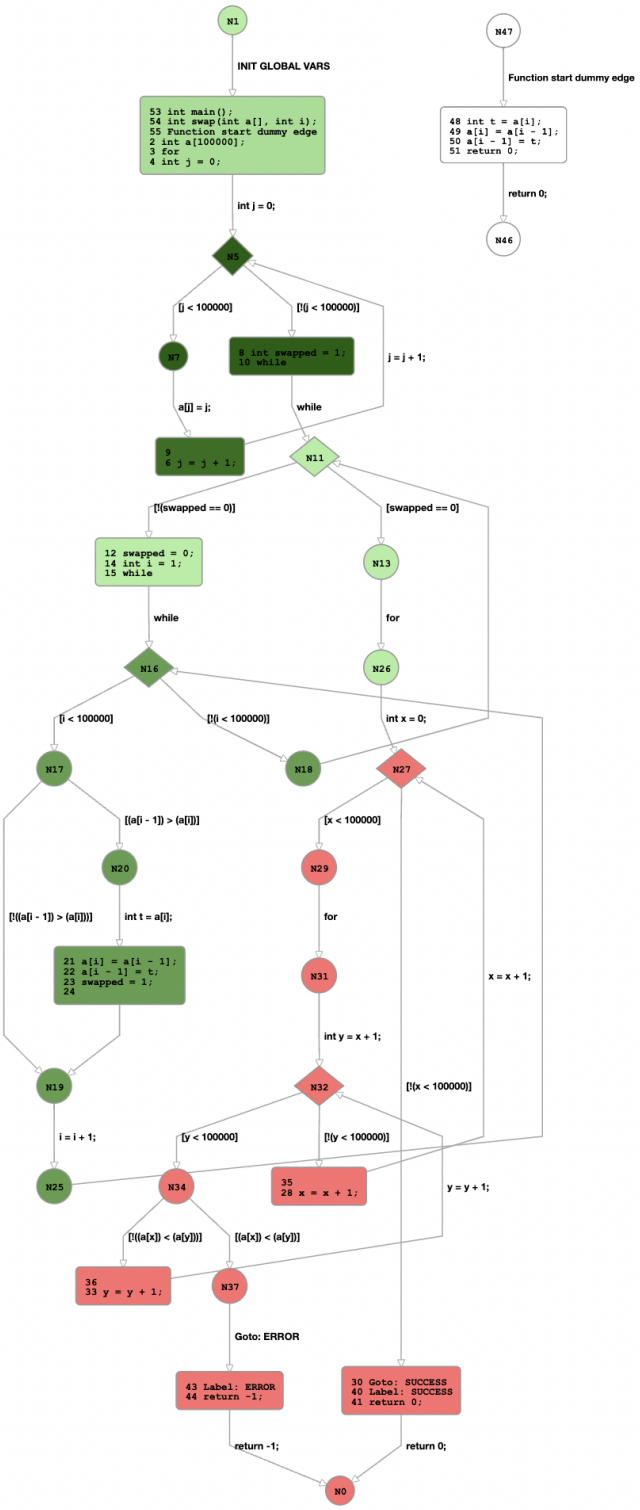


Figure 5.2: CFA with *considered-locations heat-map coloring* for a bubble sort C-program.

say that the CFA locations N46 to N51 were not visited or reached, indicating that we could have dead code fragments. Dead code in this context means that we have fragments of code lines, like an unused method, which are never considered during analysis since it is not called in the main code. For our example, this is indeed the case since we have created a helper method for the swap operation, which is actually not used.

5.2.2.2 Discussion

Considered-locations heat-map coloring is an approach that can be used in the context of completeness of properties or completeness of verification procedure. The first mentioned application purpose is supported by the visualization style regarding the unreached CFA locations. We expect that for most programs, all CFA locations should be at least visited. If this is not the case, we can detect the white marked locations within the CFA and check if they should indeed not be visited. If not, this indicates that our used specification was probably not appropriate for the verification task. Regarding completeness of verification procedure, we can detect CFA locations that were visited relatively often or not. For an unknown case where we have locations that are relatively rarely visited or marked as considered but not reached, we have an indication that the verifier was starting to focus on those locations. When we have, for example, two for loops with the same amount of iterations which we unroll sequentially one by one, and the analysis terminates at the beginning of the second one, we would be able to detect the verification progress with the help of the heat-map coloring within the CFA. Nevertheless, it is important to note that this interpretation always depends on the source code context and cannot be generalized. In addition, *considered-locations heat-map coloring* can also lose its expressiveness if we use an analysis that works with high over-approximation. This can lead to high visit rates as long as the analysis does not precise in its abstraction. One example would be predicate analysis with CEGAR. Here we have in the beginning a low precision which increases during the analysis. Therefore we have visited locations that we would later not visit due to the generation of suitable predicates, which narrows the program path choices. Since we collect all visited locations from the beginning of the analysis, our result regarding the count of visits per location is greater as if we would have started initially with a high precision. One idea to fix this issue for predicate analysis with CEGAR is our next proposed visualization approach.

5.2.3 Predicate-Considered-Locations Coloring

5.2.3.1 Idea

Predicate-considered-locations coloring uses the same *PredicateConsideredLocation* set as the one used for the *predicates-considered-locations* coverage measure. Therefore if a location is predicate considered depends on Algorithm 2. Instead of calculating a single coverage value, we now only color all locations within the CFA which are contained in the *PredicateConsideredLocation* set.

5.2.3.2 Discussion

Predicate-considered-locations coloring has the advantage of reducing, in many cases, over-approximation regarding the consideration of covered locations. For illustration, we look at Fig. 5.3, where we use predicate analysis with CEGAR for the program from Appendix A.2. We see on the left side of the figure a CFA with *predicate-considered-locations coloring*, and on the right side a CFA with *visited-locations coloring*. Note that *considered-locations heat-map coloring* would have marked the same locations as covered, with the difference that we would have a color gradient for each covered location. We nevertheless use *visited-locations coloring* for this example since it uses the same green coloring tone, which therefore simplifies this comparison. The main difference here is that all locations beginning from N13 are not covered with *predicate-considered-locations coloring*. This is due to the reason that we have from N13 leaving CFA assume edges where the corresponding code contains variables which are not contained in any predicate formula of our predicate set at this location. In this case, we have no information regarding the variable j, since we declare it at the beginning of the program, but we do not initialize it with any value. Therefore we cannot safely decide which of the if-branches could be feasible. On the other hand, the analysis visits both branches due to over-approximation. For the analysis, this behavior is expedient, but regarding the coverage, this leads to a high location coverage where in this case, every location is considered as covered. The interpretation of this visualization is almost meaningless because the only information we can deduce is that we have visited all locations at least once, meaning that every location is syntactically reachable.

The visualization approach for *predicate-considered-locations coloring* can similarly be applied for the *predicates-relevant-variables* coverage measure. The difference would be that we have fewer locations colored due to the additional variable filtering.

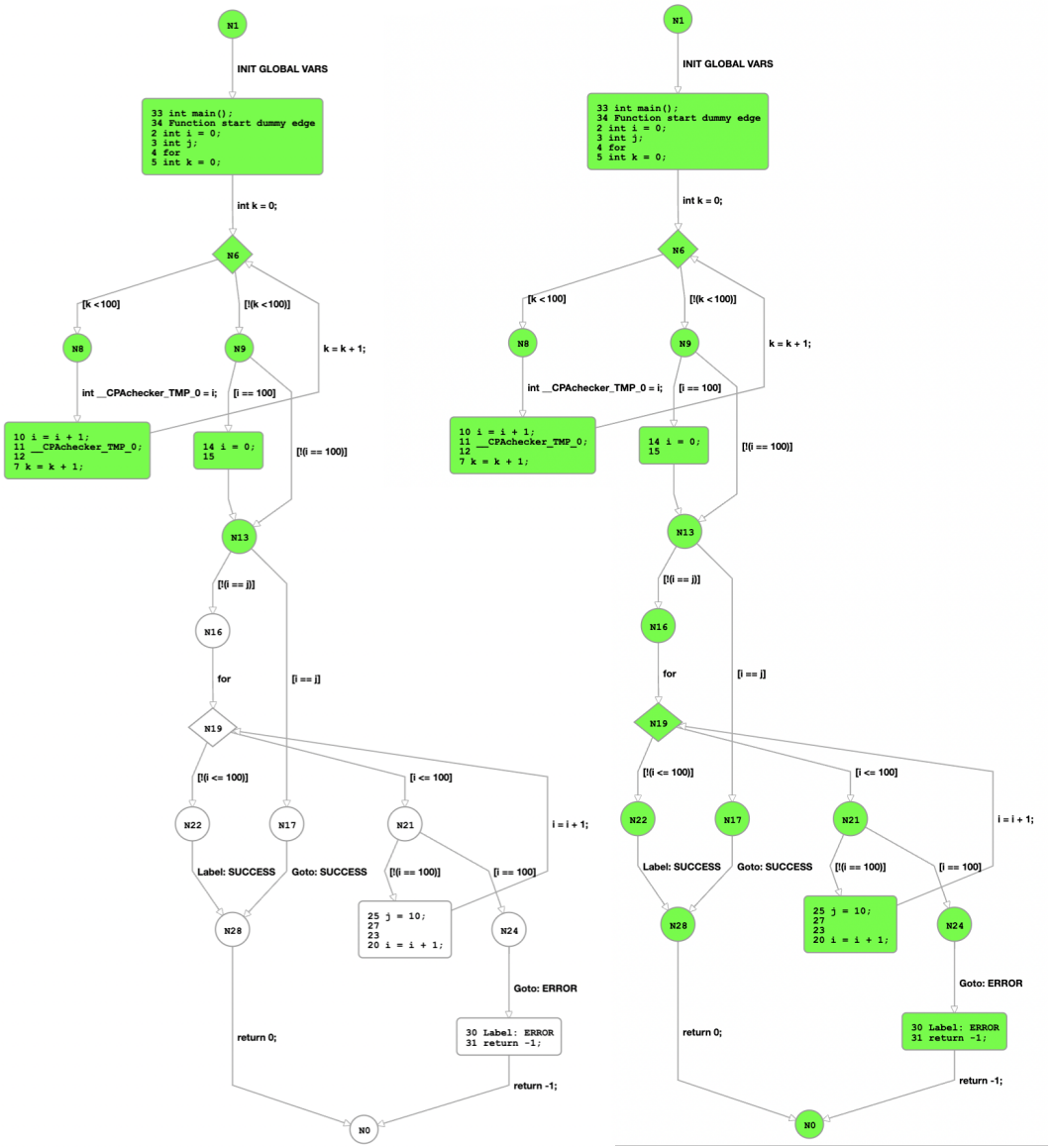


Figure 5.3: Uncompleted predicate analysis with CEGAR for the program from Appendix A.2. CFA with *predicate-considered-locations coloring* (left) and *visited-locations coloring* (right).

5.3 Program Variable Visualization

In this section, we look at visualization approaches that focus on program variables. This kind of visualization can be applied to variables occurring in code statements. Depending on our coverage criteria, we mark all variable names within a code statement with a color different from the rest of the statement text. Statements typically occur within the source code or at CFA edges. In this thesis, we limit our scope to statements of a given source code. Coloring covered variables could help the verification engineer check at the end of a finished analysis if the given specification was appropriate. This is due to the reason that having variables that were not covered indicate that we did not consider every part of the program code. In the following, we look at coverage approaches based on coloring variables. Moreover, we discuss potential benefits compared to the previous coverage visualization techniques.

5.3.1 Visited-Variables Coloring

5.3.1.1 Idea

Visited-variables coloring is based on the *visited-variables* measure approach. The difference here is that we use the *VisitedVariables* set as coverage criteria, meaning we check for each variable occurrence within the source code if the variable is contained within the *VisitedVariables* set. If the condition is fulfilled, we highlight the variable in a different color. For this kind of visualization, it is also important to consider the variable scope. This means that we also need to check if the variable we want to color matches the scope of the variable contained in the *VisitedVariables* set. Variables not contained within the set or out of scope keep their default color.

5.3.1.2 Discussion

Coloring covered variables within the source code instead of source code lines has the advantage that the outcome for different analyses varies in most cases less. This is caused by the having typically more statements than variables within source code, since every variable declaration is also a statement, and we additionally could have statements without any variables.

To illustrate our idea, we look at a C-program which we verify with the verification framework CPAchecker, one time with predicate analysis with CEGAR and the other time with the default value analysis Fig. 5.4. For the source code line visualization, we take the approach *visited-lines heat map coloring*, meaning that we mark visited lines green. We then can see for the same program and specification, two different coverage coloring results. While we

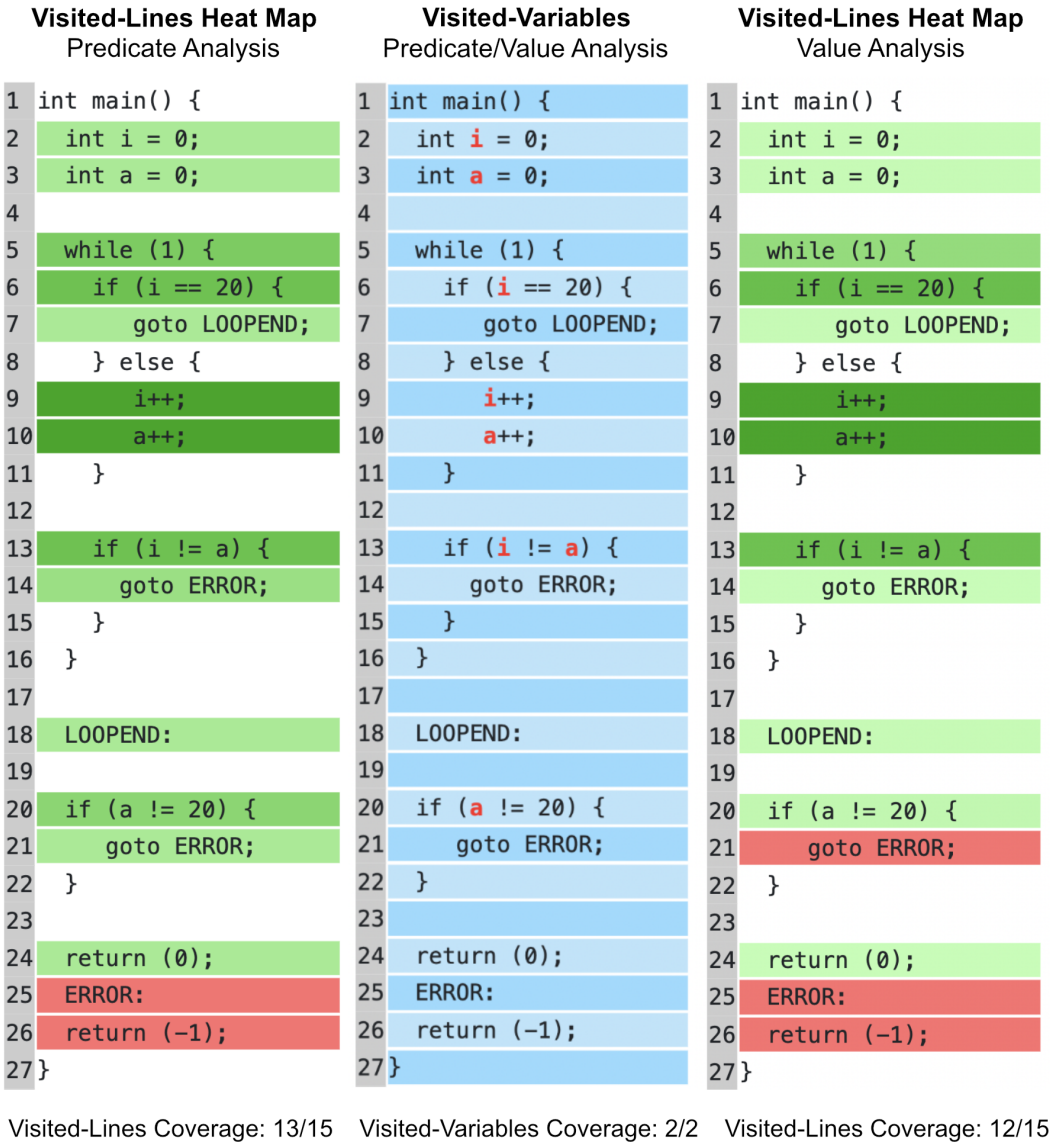


Figure 5.4: Coverage visualization comparison of the same C-program for different analyses. Note, that we have used CPAchecker as verification framework and for all predicate analysis cases we have used also CEGAR. The column in the middle is valid for predicate analysis as well as for value analysis. C-program taken from doc/examples/example.c in the CPAchecker repository (`https://gitlab.com/sosy-lab/software/cpachecker/-/blob/trunk/doc/examples/example.c`)

consider line 21 visited with predicate analysis, we consider it with value analysis as not visited. This behavior is reasoned by the initial over-approximation of the predicate analysis due to CEGAR. On the other hand, when we look at the middle column of Fig. 5.4, we see that for both analyses, the same coverage visualization results, showing us that all occurring variables were visited. This outcome facilitates the interpretation of the coverage visualization regarding the question if our used specification was suitable for our verification task since we do not need to consider each analysis case separately.

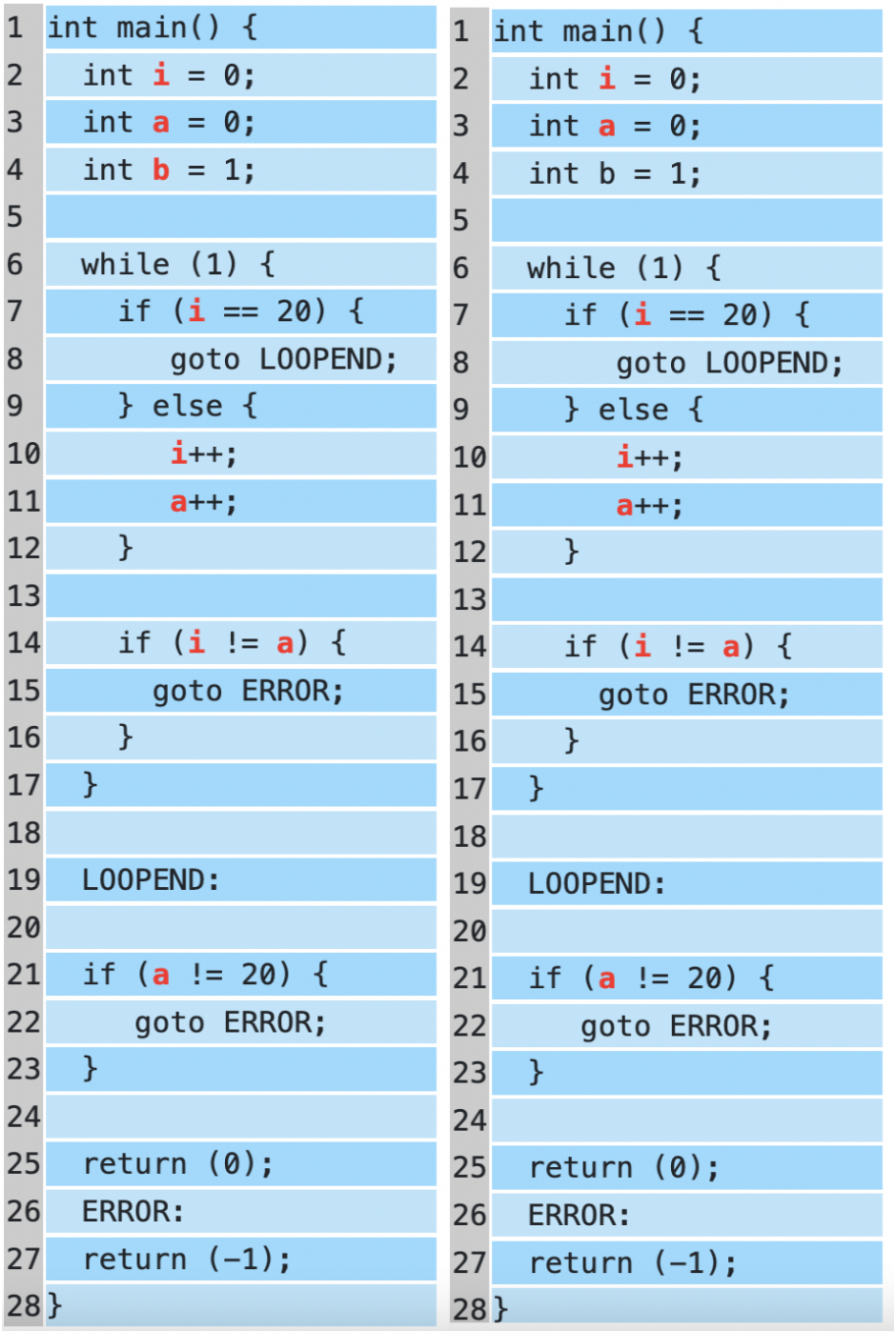
5.3.2 Predicate-Abstraction-Variables Coloring

5.3.2.1 Idea

Predicate-abstraction coloring is based on the *predicate-abstraction-variables* measure approach. This visualization approach works similar to *visited-variables coloring*, except that we use the *PredicateAbstractionVariables* set as basis for our coverage coloring. The idea of this approach is to only look at variables that are used by the verification analysis. Since *PredicateAbstractionVariables* set depends on abstraction formulas, this visualization is only applicable when we use predicate analysis.

5.3.2.2 Discussion

Compared to *visited-variables coloring*, the advantage of *predicate-abstraction-variables coloring* is that we do not color unused variables. When we now consider Fig. 5.5, we see that in line 4, in the case of *visited-variable coloring*, we highlighted the variable `b` red, meaning that this variable is covered, whereas, in the case of *predicate-abstraction-variables*, we do not color the variable`b` red. This is due to the reason that `b` is, after its declaration, not used anymore and therefore not occurring in any abstraction formula. Having this coloring behavior is preferable because the verification engineer can better interpret the verification result. For our example case (cf. Fig. 5.5), we would detect `b` as uncolored, concluding regarding the context of the whole program that `b` was unused, meaning either our program code contains unnecessarily the variable `b` or our specification was not suitable for our problem case. On the other hand, the advantage of *visited-variables coloring* is that it is analysis-independent usable. This has as consequence that we can compare the visualization for different analyses.



```
1  int main() {
2    int i = 0;
3    int a = 0;
4    int b = 1;
5
6    while (1) {
7      if (i == 20) {
8        goto LOOPEND;
9      } else {
10       i++;
11       a++;
12     }
13
14     if (i != a) {
15       goto ERROR;
16     }
17   }
18
19   LOOPEND:
20
21   if (a != 20) {
22     goto ERROR;
23   }
24
25   return (0);
26   ERROR:
27   return (-1);
28 }
```

```
1  int main() {
2    int i = 0;
3    int a = 0;
4    int b = 1;
5
6    while (1) {
7      if (i == 20) {
8        goto LOOPEND;
9      } else {
10       i++;
11       a++;
12     }
13
14     if (i != a) {
15       goto ERROR;
16     }
17   }
18
19   LOOPEND:
20
21   if (a != 20) {
22     goto ERROR;
23   }
24
25   return (0);
26   ERROR:
27   return (-1);
28 }
```

Figure 5.5: Comparison between *visited-variables coloring* (left) and *predicate-abstraction-variables coloring* (right) for the same program and analysis (predicate analysis with CEGAR). C-program, except line 4, taken from doc/examples/example.c in the CPAchecker repository (`https://gitlab.com/sosy-lab/software/cpachecker/-/blob/trunk/doc/examples/example.c`)

6 Implementation

In this chapter, we want to look at the implementation details of our verification coverage approaches and visualization methods. We use the verification framework CPAchecker[1] [6], a framework for formal program verification, to implement our work. Most of the ideas are implemented in Java. For the visualization part, we also use JavaScript as programming language. We structure this chapter regarding the dataflow. Therefore we begin discussing the implementation of the measures, going over to the collectors, and then looking at the visualization.

6.1 Verification Measures

At first, we want to look at our newly created `util.coverage.measures` package. The corresponding class diagram is depicted in Fig. 6.1. We start with a Java interface definition called `CoverageMeasure`. It consists of three basic methods which every verification coverage measure should implement. The method `getNormalizedValue()` returns a double representing the actual normalized coverage value. In addition, we want to have a `getValue()` method which returns the coverage value without normalization, representing just the count of some specific verification coverage data analyzed. This definition is similar to the Function (3.1). The difference is that we take the coverage data from the class field where this interface is implemented instead of taking a method parameter. The third interface method is `getMaxValue()` which

[1] `https://cpachecker.sosy-lab.org`

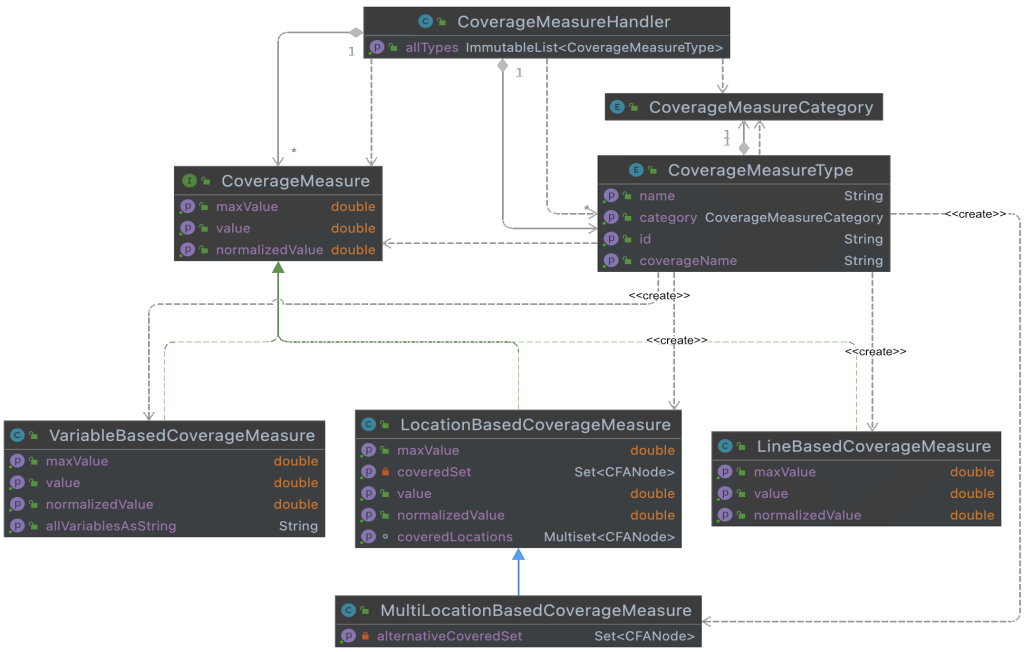


Figure 6.1: Class diagram of the measures package.

should be the theoretically maximum possible measure value, typically used for normalization of the measurement.

We cluster our measurement approaches into categories, defined in the enum `CoverageMeasureCategory`. We can have different realizations like `LocationBased` or `LineBased`, representing a category of similar coverage approaches. The idea here is to save redundant implementation work and allow an easy way to add more coverage categories in the future. Therefore we also have a `CoverageMeasureHandler` class, which holds a map where the keys are instances of `CoverageMeasureType` and the values are classes which implements the `CoverageMeasure` interface. The already mentioned `CoverageMeasureType` is an enum consisting of all of our discussed verification coverage measure approaches. This enum also defines for each entry a `CoverageMeasureCategory`. With this information, we can deduce the appropriate coverage measure implementation for each case.

6.1.1 Coverage Measure Implementations

We have implemented four different classes for the `CoverageMeasure` interface.

6.1.1.1 LocationCoverageMeasure

The `LocationCoverageMeasure` class holds a multiset of CFA locations that were covered during the analysis. The coverage criterium depends on the used `CoverageMeasureType` and is decoupled from this measure class. The coverage calculation is implemented as the quotient of the `getValue()` and `getMaxValue()` methods, where `getValue()` returns the duplicate-free set of CFA locations and the `getMaxValue()` the number of all possible locations. For later visualization purposes, a `getColor(CFANode location)` method is defined to return the coverage status, encoded as a color representation, for a given location identification number in the CFA.

6.1.1.2 LineCoverageMeasure

This implementation is similar to the `LocationCoverageMeasure` class. The main difference is that we have as coverage data a multiset of covered source code lines for each source file and a set of all source code lines considered for each file. Consequently, the previous color method `getColor(Integer location)` is now defined as `getColor(String file, int line)`, returning the coverage color for a given source file at a given source line.

6.1.1.3 MultiLineCoverageMeasure

`MultiLineCoverageMeasure` inherits from `LocationCoverageMeasure`. This class has an additional multiset of considered locations. Therefore it is possible to look at the combination of multiple coverage criteria and build, for example, an intersection of multiple coverage multisets.

6.1.1.4 VariableCoverageMeasure

As last interface implementation we discuss `VariableCoverageMeasure`. This class also has two class fields representing the coverage data. One is a multiset of all variables, the other of all relevant variables. Both use variables encoded as Strings. The coverage is defined as the quotient of the element set sizes of the multisets.

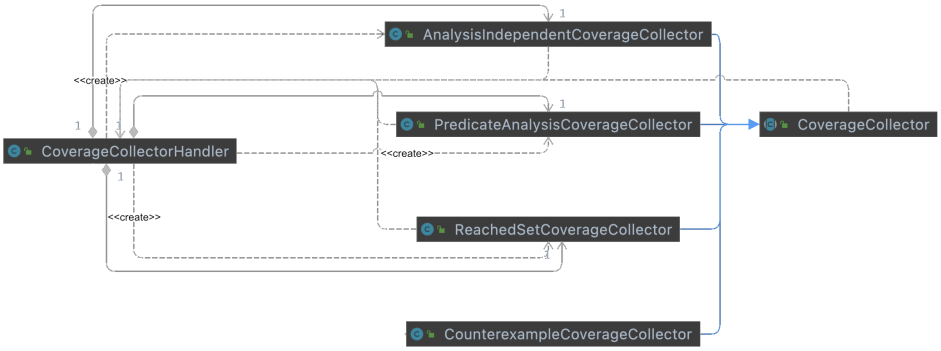


Figure 6.2: Class diagram of the collectors package.

6.2 Coverage Data Collectors

The coverage data collection is handled by the `util.coverage.collectors` package. It consists of a `CoverageCollectorHandler` which hold all different collector classes (cf. Fig. 6.2). Each collector class inherits basic collector capabilities from the abstract class `CoverageCollector` and combines it with collector-specific methods.

6.2.1 Coverage Collector Implementations

We have implemented two new classes and refactored two already existing classes to accomplish the coverage data collection.

6.2.1.1 AnalysisIndependentCoverageCollector

The `AnalysisIndependentCoverageCollector` can be used during any kind of verification analysis. It has methods for adding new visited locations or lines. This class also holds references to a `TimeDependentCoverageHandler` instance. The collector needs this instance to start the proper initialization of the TDC. We have implemented a `collect()` method, which is called at the end of the analysis. It is used to populate the coverage data for all initialized measures.

6.2.1.2 PredicateAnalysisCoverageCollector

The `PredicateAnalysisCoverageCollector` class is designed similar like the `AnalysisIndependentCoverageCollector`. The difference is that it is used only for predicate analysis. Therefore this collector has many methods which adds predicate analysis related coverage data to the set of `CoverageStatistics`.

6.2.1.3 ReachedSetCoverageCollector

In contrast to the two mentioned collectors, `ReachedSetCoverageCollector` already existed in CPAchecker. It was slightly redesigned and renamed to fit better into the collector structure. Hence, we remove the old coverage data reference and add all relevant coverage data to local class fields, which are later used for all initialized measures. This new design benefits from a smaller coupling between the package components. We use at the end of the analysis the `ReachedSetCoverageCollector` when we have a reached set as the basis for the coverage collection.

6.2.1.4 CounterexampleCoverageCollector

In analogy to the `ReachedSetCoverageCollector` the same refactoring applies for the `CounterexampleCoverageCollector` class. The difference here is that we use a counter example path instead of a reached set for coverage data collection. This more modular design has the advantage that if we want to use an alternative implementation of this kind of coverage collector, we do not need to change every occurrence of this collector in code, instead we only need to switch the returned coverage collector instance within the `getCounterexampleCoverageCollector()` method which is defined in `CoverageCollectorHandler`.

6.2.2 Adding New Measures

When we want to add new measures in the future which belong to one of the already discussed coverage categories, we need to define a new enum case in `CoverageMeasureType` and extend the `getCoverageMeasure()` method by the new case. This is needed to specify which coverage data we want to consider for this measure. In addition, we need to locate the right child class of `CoverageCollector` and add methods for coverage data gathering and retrieving. Everything else regarding the visualization and data processing is automatically handled by the package.

6.3 Coverage CPAs

Next, we want to look at the `cpa.coverage` package. We redesign this package to adapt to the already proposed structure. To start with we have a `CoverageCPA` class which is responsible for the coverage collection. It has a method which is called `getCoverageCollectorHandler()`, which returns an instance of the already discussed `CoverageCollectorHandler`. This handler instance is initialized at the beginning of the analysis and is passed to the `CoverageCPA` via reflection.

6.3.1 Analysis-Independent

This class implements the `ConfigurableProgramAnalysis` interface, which is used for all CPAs in CPAchecker. It is the realization of the discussed four-tuple CPA from Chapter 3.3.1. We also have a `CoverageTransferRelation` class which implements the `TransferRelation` interface. Consequently, we can customize the implementation of the `getAbstractSuccessorsForEdge` method, which gets called by the CPA core algorithm. Within this method, we use the proper coverage collector instance from `CoverageCollectorHandler` instance to gather suitable coverage data during the transfer relation step. In addition, we can track the elapsed time between the last data collection, which is later used by the time-dependent coverage graph (TDCG).

6.3.2 Predicate Analysis

In contrast to the `CoverageCPA` class approach we do not need an analogous implementation, since we can use the `PredicateCPA` and extend it by a `CoverageCollectorHandler` instance which is passed from the beginning of the analysis. If we specify in the configuration that we want to collect predicate-analysis-related coverage data we return within the existing method `getTransferRelation()` an instance `PredicateCoverageTransferRelation` instead of `PredicateTransferRelation`. We define this new transfer relation class analogously to the `CoverageTransferRelation` class. The difference here is that we have specific processing methods to extract coverage-related data, which the `PredicateAnalysisCoverageCollector` instance can then collect. For example we have defined a `processPredicates()` method in the `PredicateAnalysisCoverageTransferRelation` class which extracts all predicates for a certain location and passes it a suitable collector.

6.4 Time-Dependent Coverage Graph (TDCG)

The TDCG data processing is handled by the `util.coverage.tdcg` package. Here we have `TimeDependentCoverageData` class which holds the actual data as a map. Since we can have different coverage criteria, we have multiple TDCGs we need to distinguish from each other. To accomplish this we have a `TimeDependentCoverageType` enum listing all types of TDCGs. The `TimeDependentCoverageHandler` class builds a connection between each `TimeDependentCoverageType` entry to a `TimeDependentCoverageData` instance. It is responsible for initializing data and works from outside as access point to retrieve the appropriate `TimeDependentCoverageData` instance.

6.5 Statistics Report

We can specify in CPAchecker the option to generate a `Report.html` file when the analysis is done. This can be useful for evaluating the analysis process. We have adapted the already existing `CoverageReportStdoutSummary` class to our new package structure. Its main purpose is to print all coverage data to the standard output. It is also used to print for each of our coverage measures some statistics. Those are displayed in the `Code Coverage` section in the Statistics tab. Therefore it is possible to read with the help of an internet browser the overall coverage value for each measure or the count value of its coverage data.

6.6 Visualization

The visualization of our approaches is realized, similar to the statistics report. When the analysis is done, the collected data is processed in the already existing `ReportGenerator` to a JSON object which is used as part of the input data for the `report.js` JavaScript file. The HTML report, which depends on `report.js` has an additional menu tab item if TDC data exists and depicts the TDCG with the help of the `D3` JavaScript library. `D3.js` is used for manipulating documents based on data[2]. In addition, we have implemented a selector within the TDCG tab view to allow the user to choose between different coverage measures as the basis for the TDCG. We have created the whole TDCG tab as part of this thesis. An example of how it looks is depicted in Fig. 6.3. We also implemented a coverage HTML selector for the Source tab. Every coverage measure based on the `LineCoverageMeasure` is

[2] `https://d3js.org`

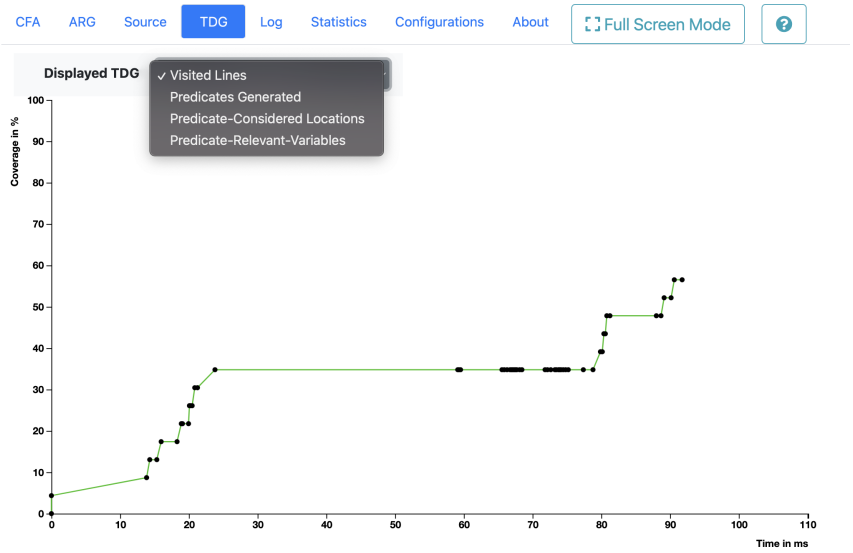


Figure 6.3: Screenshot of the report.html where the TDG tab is displayed with an applied coverage visualization.

automatically shown as a visualization option within that selector. Depending on the implemented coverage criteria, the background of the source code lines is colored when an option is selected. As an alternative approach, we can also use coverage measures based on the `VariableCoverageMeasure` class to color relevant variables occurring in the source code text. This kind of visualization can also be selected within the Source tab in the same selector. As third visualization approach we have implemented a selector for the CFA tab (cf. Fig. 6.4). Depending on the selected coverage option, the nodes of the CFA graph are colored depending on their coverage status, which is defined in the `LocationCoverageMeasure` class. Since we use multisets for the covered locations or source code lines, there is also support for coverage heat maps. To accomplish this, we deduce the relative frequency of each location or line by counting all the same occurrences and dividing it by the count of the element with the highest occurrence. Therefore we can use this relative frequency, which is a number between 0 and 1, and build a gradient of two colors. When we look at Fig. 6.5, we see how this would look like for the Source tab. All mentioned selector options are automatically filled with entries that were relevant for the analysis. This means that if we do predicate analysis, there are only visualization options for analysis-independent coverage approaches and predicate analysis-specific coverage approaches but none for any other analysis category.

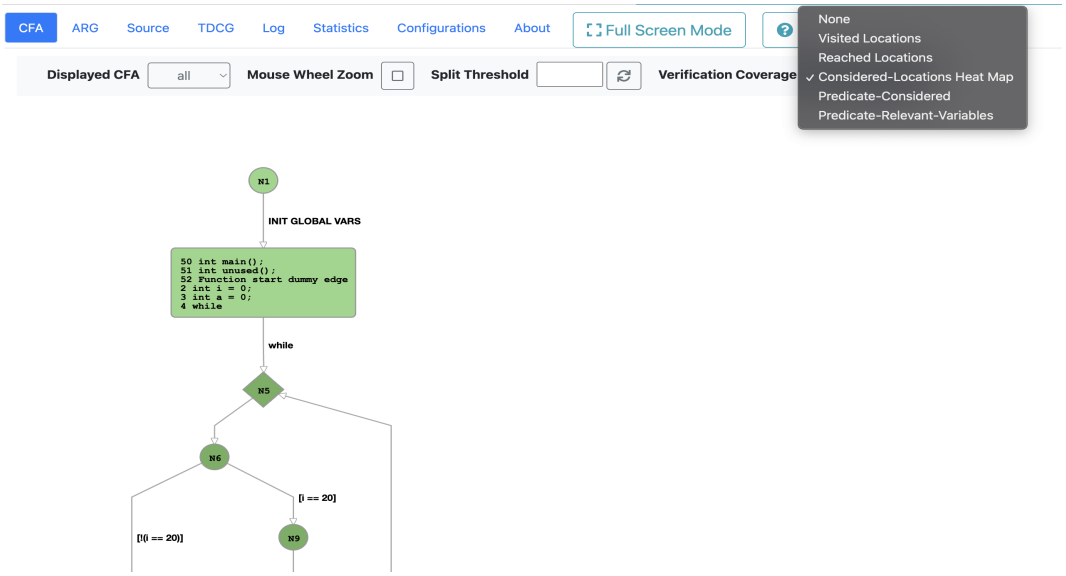


Figure 6.4: Screenshot of the report.html where the CFA tab is displayed with an applied coverage visualization.

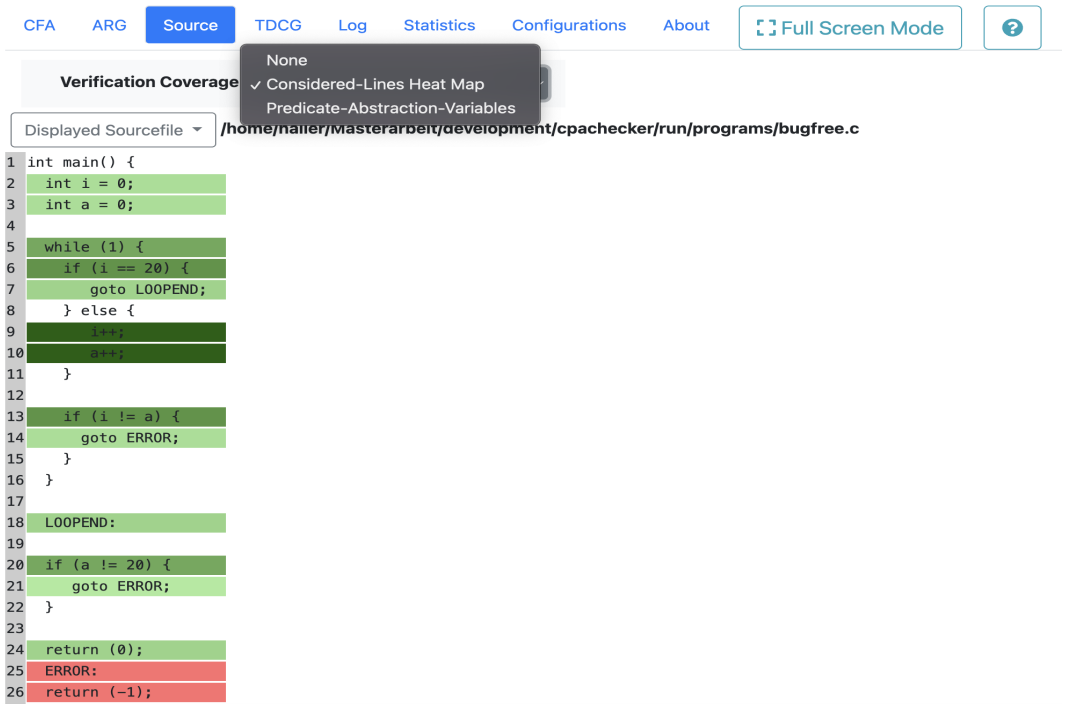


Figure 6.5: Screenshot of the report.html where the Source tab is displayed with an applied coverage visualization.

7 Evaluation

We want to compare the proposed verification coverage approaches to each other on different programs and for different verification analyses. We also show how *time-dependent coverage graphs* (TDCGs) look like for different program scenarios. Moreover, we benchmark the additional time it needs to calculate the coverage.

7.1 Setup

As verification framework we use CPAchecker revision `r40819`. The execution system has a Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz CPU with 16GB of RAM. The installed operating system is Ubuntu 20.04.4 LTS.

We select C-programs from SV-COMP 2022[1] [2]. The used specification is given as a property file and consists of one line that includes the entry function and a linear temporal logic (LTL) property. We take regarding LTL the basic definitions from the literature [7]. More precisely, our specification uses a `main()` function as entry point and checks if never a *reach error* occurs:

$$\text{CHECK(init(main()), LTL(G ! call(reach\_error())))} \tag{7.1}$$

A reach error only happens if a feasible path in the CFA contains a CFA edge with `reach_error()` as corresponding code. This `reach_error()` function call is contained at locations which should never be reached during program execution. We make use of this function in all of our programs for evaluating our

[1] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`

verification coverage measures. In the following, we list all of these programs regarding their category:

- ReachSafety-Arrays
  - array_doub_access_init_const.c (arra)
  - brs1f.c (brs1)
  - zero_sum_const1.c (zero)
- ReachSafety-BitVectors
  - diamond_2-1.c (diam)
  - jain_5-2 .c (jain)
  - num_conversion_2.c (num_)
- ReachSafety-ControlFlow
  - kbfiltr_simpl1.cil.c (kbfi)
  - s3_srvr_1b.cil.c (s3_s)
  - test_locks_7.c (test)
- ReachSafety-Loops
  - multivar_1-1.c (mult)
  - nested_1b.c (nest)
  - simple_vardep_1.c (simp)
- SoftwareSystems-DeviceDriversLinux64-ReachSafety
  - 32_1_cilled_ok_nondet_linux-3.4-32_1-drivers–acpi–bgrt.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.i (32_1)
- SoftwareSystems-BusyBox-ReachSafety
  - dirname-1.i (dirn)

Note that we will use in the following an abbreviation for each program which consists of the first four characters and acts as an identifier. Regarding the program choice, we limit our scope to ones which are suitable for reachsafety analyses. We select programs depending on different statistics, like some with relatively many conditions or some with many code lines (cf. Table 7.1). The idea is to show the resulting coverage values for different common scenarios.

Table 7.1 Overview of C-programs for our evaluation. In the expected verdict column we also put the result of our evaluation in parentheses if it differs.

Programs	Total Lines	Total Functions	Total Conditions	Total Variables	Total Locations	Expected Verdict
arra	13	4	8	3	47	true (unknown)
brs1	28	4	14	4	63	false
zero	24	4	14	5	62	true (unknown)
diam	28	3	24	2	83	false
jain	11	3	2	2	32	true (unknown)
num_	17	3	6	5	40	true
kbfi	380	12	82	56	399	true
s3_s	53	2	30	4	87	true
test	65	2	44	15	126	true
mult	10	3	4	2	37	true
nest	8	2	4	1	8	false
simp	12	3	4	3	33	true
32_1	2960	35	48	222	1024	true
dirn	585	17	68	186	832	true

7.2 Coverage Measure Overview

Before we show our results, we briefly want to give an overview of the measures which we want to evaluate in Table 7.2. Note that we have shortened the terms completeness of properties to *property* and completeness of verification procedure to *verification procedure*. The column "Input Type" describes for each coverage measure the kind of data the given input set is based on. Line-based means we use covered source code lines as input, variable-based means we look for covered program variables, and location-based analogously stands for covered CFA locations.

Table 7.2 Overview of all mentioned verification coverage measures.

Measure	Analysis Type	Application Purpose	Input Type	Output Domain
Visited-Lines	independent	property	line-based	ratio
Visited-Variables	independent	property	variable-based	ratio
Predicate-Abstraction-Variables	predicate analysis	property	variable-based	ratio
Predicate-Considered-Locations	predicate analysis	verification procedure	location-based	ratio
Predicate-Relevant-Variables	predicate analysis	verification procedure	location-based	ratio

7.3 Program-Comparison of Measures

When comparing all proposed coverage measures for all mentioned programs, we see in Table 7.3 that *visited-variables* coverage has, in most cases, the highest value. It is often 100 %, meaning that we have considered every variable within the program as covered. For the two programs of the category SoftwareSystems, we see that *visited-variables* coverage is less than 100 %. In case of the program `dirname-1.i` we have a coverage value of 96 %. Since it is not 100 %, this is an indication that possibly the program is not completely considered by the verifier. Consequently, we check in addition the *visited-variables* coverage coloring for this program. Here we can indeed spot variables that are not marked covered. For example some within the method `vasprintf()`. Further investigations lead to the discovery that this method is never called, wherefore we can conclude that the verifier indeed did not verify the whole code for reachsafety errors. Suppose we would add as the first statement within this method a `reach_error()` function call so that if one uses this method, it would always fail, then the verifier will not detect this property violation since it was never called during the main routine. With this information, the verification engineer can now optimize this case.

When we compare *visited-variables* coverage to *predicate-abstraction variables* coverage we see that its value is always greater or equal. This behavior is expected since we now also check if the variables were used within predicate abstraction formulas to rule out potentially unused variables. This approach,

Table 7.3 Coverage values for all proposed measures for each mentioned program from Table 7.1.

Programs	Visited-Lines	Visited-Variables	Predicate-Abstraction-Variables	Predicate-Considered-Locations	Predicate-Relevant-Variables
arra	85 %	100 %	67 %	45 %	45 %
brs1	96 %	100 %	50 %	21 %	21 %
zero	67 %	100 %	60 %	23 %	23 %
diam	96 %	100 %	50 %	28 %	28 %
jain	91 %	100 %	100 %	75 %	75 %
num_	88 %	100 %	60 %	70 %	70 %
kbfi	99 %	100 %	0 %	25 %	22 %
s3_s	94 %	100 %	75 %	22 %	22 %
test	97 %	100 %	0 %	26 %	26 %
mult	80 %	100 %	100 %	78 %	78 %
nest	100 %	100 %	100 %	86 %	86 %
simp	83 %	100 %	100 %	76 %	76 %
32_1	96 %	81 %	0 %	58 %	58 %
dirn	90 %	96 %	0 %	66 %	66 %

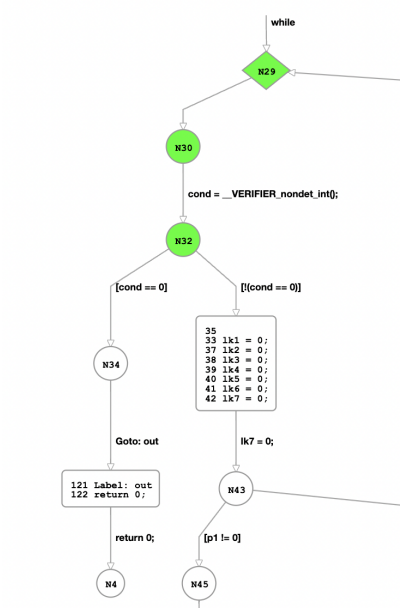


Figure 7.1: Fragment of the CFA from the C-program test_locks_7.c which shows the issue with predicate-considered locations coverage.

nevertheless, has some outliers where the actual value is 0 %. This is due to the reason that for some programs, we have no variables which we can extract from the predicate abstraction formula. These cases show that here the *predicate-abstraction variables* coverage measure is not suitable since it does not deliver helpful information.

We also can see that the coverage values of *predicate-considered locations* are in most cases less than for the previously mentioned measures. To better understand the reason behind this, we compare the program categories Reachsafety-Loops and Reachsafety-ControlFlow for this measure. We can detect for this comparison a general problem of *predicate-considered locations* coverage. The values for our programs in Reachsafety-ControlFlow are much lower than those for Reachsafety-Loops. This is due to the reason that when we reach during the program analysis a branch within the CFA, there is the possibility that we do not consider the following nodes as covered as we see, for example, in Fig. 7.1. In detail, we see in this CFA that we do not mark any locations as covered after N32, since we have no predicate formula containing the variable `cond`. This leads to the behavior that we do not consider all following locations as covered, since their previous location was not considered covered. On the other hand, we have many conditions in the Reachsafety-ControlFlow category, wherefore we have a higher chance that we early reach a branch where we do not have the variables of the condition contained in any predicate formula and therefore exclude the following program paths. This path exclusion leads to a generally lower coverage value. We want to note, that this issue applies not for all programs, ideally we get throughout the analysis new predicates so that we can reconsider uncovered locations as covered.

For *predicate-relevant variables* coverage, we have used a frequency threshold of 75 %, meaning that when we sort our set of variables depending on their occurrence count in predicate formulas, we only consider the top 75 %. Nevertheless, we have for almost all program cases similar coverage values like for *predicate-considered locations* coverage. This means that the lowest 25 % of variables did not make a significant difference regarding the coverage of program locations.

Table 7.4 Coverage values for predicate analysis and value analysis for all analysis-independent measures for each mentioned program from Table 7.1.

Programs	**Visited-Lines (predicate)**	**Visited-Lines (value)**	**Visited-Variables (predicate)**	**Visited-Variables (value)**
kbfi	99 %	94 %	100 %	100 %
s3_s	94 %	91 %	100 %	100 %
nest	100 %	88 %	100 %	100 %
test	97 %	97 %	100 %	100 %
32_1	96 %	96 %	81 %	81 %
dirn	90 %	90 %	96 %	96 %

7.4 Analysis-Comparison of Measures

In this section, we want to compare our analysis-independent coverage measure approaches, when we vary the analysis type. Therefore, we compare the *visited-lines* coverage measure and *visited-variables* coverage measure applied to programs that were analyzed with predicate analysis and value analysis. Note that for this comparison, we have excluded the cases where we had an unknown result with value analysis so that we have for both cases the same outcome and therefore we can better compare it.

We see in Table 7.4 that for *visited-lines* coverage we have slight variations. On the other hand, *visited-variables* coverage is more stable between the two analyses, which is preferable for an analysis-independent approach. In detail, we see that for all of our programs in Table 7.4, we have the same coverage values. Consequently, *visited-variables* coverage facilitates the comparison of coverage values between different programs independently from the used analysis type.

7.5 Comparison of TDCGs

We want to compare the resulting time-dependent coverage graphs (TDCGs) for three of the already mentioned programs. The goal of this section is to give an overview how different verification coverage measures develop over time.

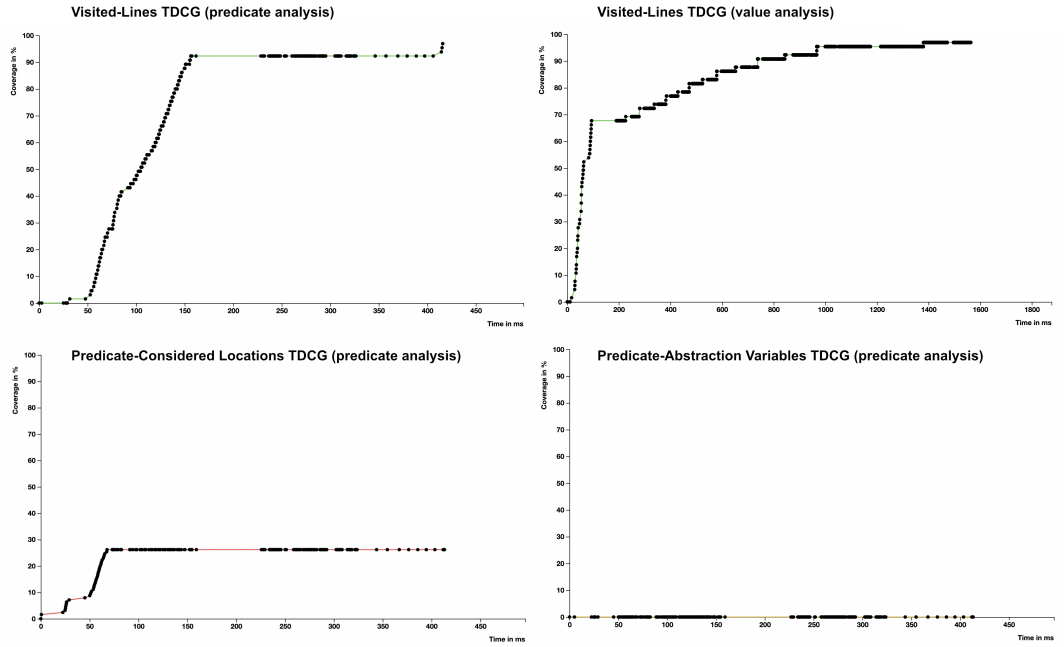


Figure 7.2: Comparison of resulting TDCGs for the programm **test**.

7.5.1 TDCGs for test_locks_7.c

When comparing the top left and top right TDCG (cf. Fig. 7.2), we see that for the first 200 ms both coverage values have a relatively fast growth. Afterward, the coverage for the predicate analysis stagnates for a long time until the analysis finishes. On the other hand, the *visited-lines* coverage for value analysis increases continuously until the end. This indicates that in the case of value analysis, we progress slower regarding the visits of new code lines. As expected, the *predicate-considered locations* TDCG is giving us a lower coverage value since this approach tries to compensate the over-approximation of considered locations. We also see that this coverage stagnates early, meaning that we get no new location that we consider covered. For the bottom right TDCG, we see a flat line where the coverage is at 0 % throughout the analysis. This is due to the reason that we always have *true* as predicate abstraction formula, which does not contain any variable. Therefore, this TDCG gives us almost no information except that we did not have any predicate-abstraction variables at all.

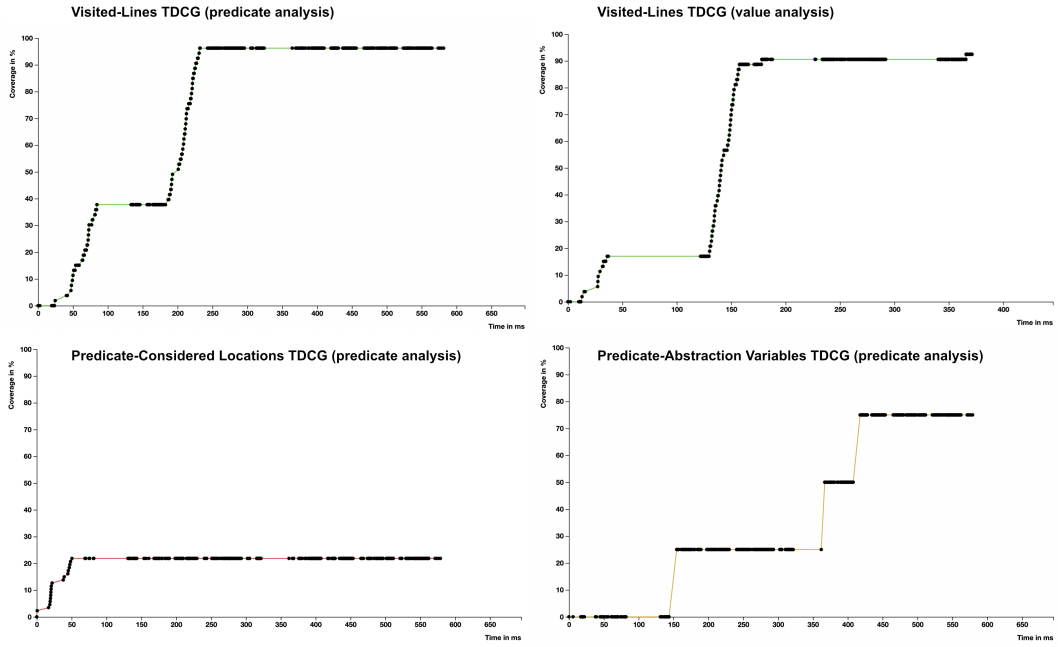


Figure 7.3: Comparison of resulting TDCGs for the program **s3_s**

7.5.2 TDCGs for s3_srvr_1b.cil.c

We start again by comparing both top-located TDCGs in Fig. 7.3. This time both graphs have a similar development. The only remarkable difference here is that the first coverage stagnation level starts for predicate analysis at about 40 %, whereas for value analysis at about 20 %. Again for the *predicate-considered locations* TDCG, we have, in comparison, a lower coverage value, which also stagnates early beginning after 60 ms. The *predicate-abstraction variables* TDCG shows clearly at which point a new single predicate-abstraction variable was considered as covered. It behaves like a step function, where each step represents a new variable.

7.5.3 TDCGs for nested_1b.c

For this program, we can see a fast growth of coverage for the *visited-lines* coverage (for predicate analysis). This is due to the low initial precision, which leads to an over-approximation of the visited states leading to many visited lines. When we compare this to predicate-considered locations TDCG, we see that with this approach we have a slower growth of coverage. Both coverage measures start stagnating after 170 ms. The *predicate-abstraction*

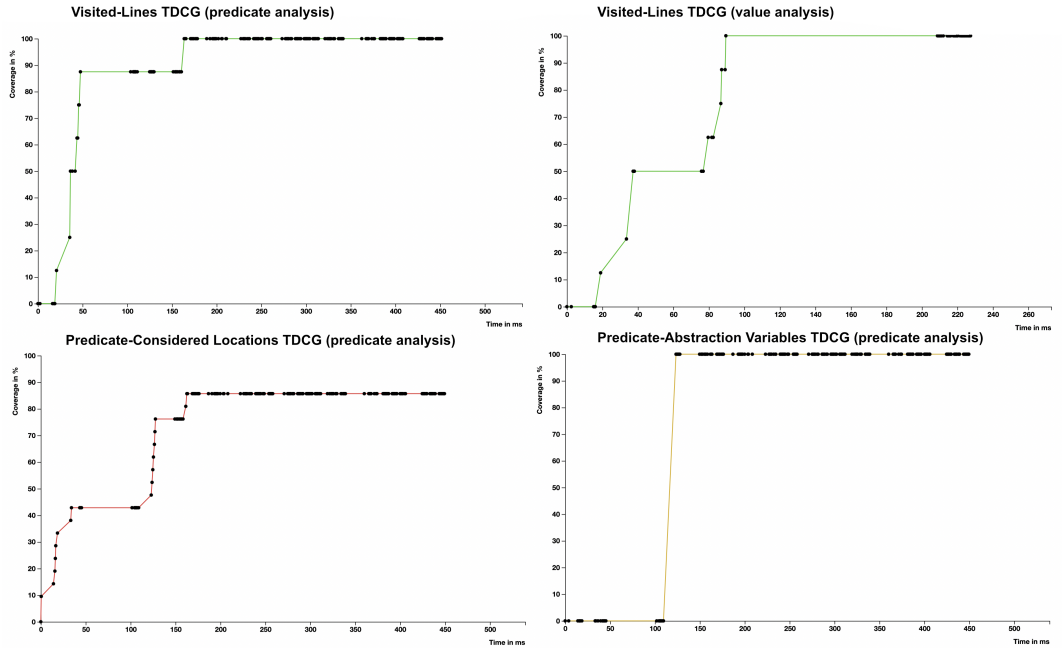


Figure 7.4: Comparison of resulting TDCGs for the program **nest**.

variables TDCG delivers in this example not much information since we only have one variable for this program, and at that point where we have visited this variable, we see no change in coverage. This is a good example that shows that for programs with a relatively low amount of variables the *predicate-abstraction variable* TDCG is not helpful.

7.5.4 Critical Reflection

We have seen that there is no universal TDCG solution that is useful for all program cases. *Visited-lines* TDCG has the problem of fast over-approximation. This issue is in the case of value analysis decreased, as we have seen. *Predicate-considered locations* TDCG starts stagnating in its value relatively early. Therefore, this approach has the potential to be optimized by finding a solution so that it reconsiders program paths within the CFA, which were excluded in the beginning. We did not show *predicate-relevant-variables* TDCG for our examples since it behaves similar to *predicate-considered locations* TDCG with the difference that it covers fewer locations. *Predicate-abstraction variables* TDCG can be helpful if our program does not have too few variables. Therefore we can say that the choice regarding appropriate TDCGs depends on the program. For example, visited-lines TDCG is a suitable choice for programs

belonging to the ControlFlow category where we use an analysis type that does not lead to an over-approximation regarding the visits of new lines. For programs that consist of many loops with a high iteration count, all TDCGs have issues since we then cover the same lines, variables, or locations multiple times but do not progress regarding covering new lines, variables, or locations. In this case, a visualization approach like visited-lines heat map coloring or considered-locations heat map coloring is more helpful since they would color lines or locations which were relatively often visited.

7.6 Performance Costs

We want to clarify the additional performance costs of all proposed coverage measures. Therefore, we benchmark the total CPU time of a set of C-programs that belong to the SV-COMP 2022 Reachsafety category. In detail, we have considered all programs from the following subcategories:

- ReachSafety-Arrays
- ReachSafety-BitVectors
- ReachSafety-ControlFlow
- ReachSafety-ECA
- ReachSafety-Floats
- ReachSafety-Heap
- ReachSafety-Loops
- ReachSafety-ProductLines
- ReachSafety-Recursive
- ReachSafety-Sequentialized
- ReachSafety-XCSP
- ReachSafety-Combinations

In total, we have 5400 programs. For determining the additional time, we use the benchmarking framework BenchExec[2] and for distributing the workload

[2] `https://github.com/sosy-lab/benchexec`

on multiple computers, we use the VerifierCloud[3]. CPAchecker contains a `benchmark.py` script where we can set as input an XML file which specifies which programs we want to verify. We set within this run definition also further options like the time limit of 900 seconds, CPU core count of 2, and a memory limit of 15 GB. We then can run the verification tasks with the specified options on a cluster consisting of 168 nodes. Each node has an Intel Xeon E3-1230 v5 processor with 8 CPU cores at 3.40 GHz each and 32 GB RAM. For the reference run, we specify to use predicate analysis with CEGAR and turn any coverage collection mechanics off, whereas we turn all options on for the run where we collect coverage data. The corresponding options, which are triggers coverage collection and processing, are the following:

- `-setprop shouldCollectCoverageAfterAnalysis=<boolean>`
- `-setprop shouldCollectCoverageDuringAnalysis=<boolean>`

Note that for coverage collection, we also need to specify the `CoverageCPA` which we have discussed in the Implementation chapter.

7.6.1 Results

The reference run needed a total CPU time of 2310000 seconds. The run with active coverage collection needed a total CPU time of 2390000 seconds. Therefore the additional time of coverage collection is 80000 seconds (needs 3.5 % more time). The increase of runtime is not significant, wherefore it is suitable to be turned on per default for reachsafety analyses.

[3] `https://gitlab.com/sosy-lab/software/verifiercloud`

8 Conclusion and Future Work

We have shown multiple verification coverage measure approaches and classified them in our literature overview alongside the broad spectrum of verification coverage definitions. Our goal was to help the verification engineer better understand how much of the given program was already processed and covered by the verifier. We have seen that there is no only solution to this, instead it depends on the use case, for example, which analysis we use. We also separated our ideas depending on the application purpose. Regarding completeness of properties, we have shown that it can help to detect program analyses that potentially have used an unsuitable specification for the problem case. For completeness of verification procedure, we have seen that it is difficult to work with just a single coverage number. Therefore we have focused our work on appropriate visualizations like heat maps or time-dependent coverage graphs (TDCG). Especially the last approach helps to show the verification engineer at which point the growth of the used coverage data stopped or was highest. Furthermore, the implemented TDG visualization offers not only the capability to depict coverage-related graphs, it is also suitable for other potentially interesting verifier-related data to be visualized.

The newly implemented coverage package in CPAchecker allows conveniently adding further measures, which can be printed to the stdout or visualized within an HTML report. Future research could go in the direction of further improving the proposed measures. For example, finding better heuristics to consider variables as relevant for the measure predicate relevant variables coverage. There is also room for research for further coverage measures belonging to other analysis types, like value analysis, which could give more helpful coverage information than an analysis-independent approach.

A Implementations

All project relevant code snippets are listed here chronologically in the order I have used them.

A.1 Bubble Sort C-Program

C-program which shows a bubble sort implementation[1]. The code snippet basis is taken from `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/array-examples/sorting_bubblesort_ground-2.i`.

We added two additional goto calls to indicate if the algorithm was successful or not. We also have an additional unused `swap` helper method. In the beginning we fill an array with integer values. Then the array elements should be sorted in ascending order. In the end there is a check if the next element within the array which should be sorted is greater then the previous element.

```
// SPDX-FileCopyrightText: The SV-Benchmarks Community
//
// SPDX-License-Identifier: Apache-2.0
int main() {
  int a[100000];

  for (int j = 0; j < 100000; j++) {
    a[j] = j;
  }

  int swapped = 1;
  while (swapped) {
    swapped = 0;
```

[1] `https://gitlab.com/sosy-lab/software/cpachecker/-/blob/verification-coverage/test/programs/simple/arrays/bubblesort.c`

```
    int i = 1;
    while (i < 100000) {
      if (a[i - 1] > a[i]) {
        int t = a[i];
        a[i] = a[i - 1];
        a[i-1] = t;
        swapped = 1;
      }
      i = i + 1;
    }
  }

  for (int x = 0; x < 100000; x++) {
    for (int y = x + 1; y < 100000; y++) {
      if (a[x] < a[y]) {
        goto ERROR;
      }
    }
  }
  goto SUCCESS;

  SUCCESS:
    return 0;
  ERROR:
    return -1;
}

int swap(int a[], int i) {
  int t = a[i];
  a[i] = a[i - 1];
  a[i-1] = t;
  return 0;
}
```

A.2 C-Program for Predicate-Considered-Locations Coloring Comparison

C-program to show an use case for predicate-considered locations coloring.

```
int main() {
  int i = 0;
  int j;

  for (int k = 0; k < 100; k++) {
     i++;
  }

  if (i == 100) {
    i = 0;
  }

  if (i == j) {
    goto SUCCESS;
  }

  for (; i <= 100; i++) {
    if (i == 100) {
      goto ERROR;
    } else {
      j = 10;
    }
  }

  SUCCESS: return 0;
  ERROR: return -1;
}
```

Bibliography

[1] Dejanira Araiza-Illan, David Western, Anthony Pipe, and Kerstin Eder. Coverage-driven verification —. In Nir Piterman, editor, *Hardware and Software: Verification and Testing*, pages 69–84, Cham, 2015. Springer International Publishing.

[2] Dirk Beyer. Progress on software verification: Sv-comp 2022. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 375–402. Springer, 2022.

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

[4] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: a technique to pass information between verifiers. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 57. ACM, 2012.

[5] 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.

[6] 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.

[7] Thomas Bunk. LTL software model checking in CPAchecker, 2019.

[8] Rodrigo Castaño, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. Verification coverage. *CoRR*, abs/1706.03796, 2017.

[9] Ivan Chajda, Radomír Halaš, and Jan Kühr. *Semilattice structures*, volume 30. Heldermann Lemgo, 2007.

[10] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for formal verification. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, pages 111–125, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[11] 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, sep 2003.

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

[13] Daniel Große, Ulrich Kühne, and Rolf Drechsler. Estimating functional coverage in bounded model checking. In Rudy Lauwereins and Jan Madsen, editors, *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*, pages 1176–1181. EDA Consortium, San Jose, CA, USA, 2007.

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, jan 2002.

[15] M. A. Horowitz and R. C. Ho. Validation coverage analysis for complex digital designs. In *Computer-Aided Design, International Conference on*, page 146, Los Alamitos, CA, USA, nov 1996. IEEE Computer Society.

[16] K.A. Semendjajew I. N. Bronstein. *Taschenbuch der Mathematik*. Europa-Lehrmittel, 10 edition, 2016.

[17] Charalambos Ioannides and Kerstin I. Eder. Coverage-directed test generation automated by machine learning – a review. *ACM Trans. Des. Autom. Electron. Syst.*, 17(1), jan 2012.

[18] Michael Katrowitz and Lisa M. Noack. I'm done simulating; now what? verification coverage analysis and correctness checking of the dec chip 21164 alpha microprocessor. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 325–330, New York, NY, USA, 1996. Association for Computing Machinery.

[19] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, apr 1999.

[20] William KC Lam. *Hardware design verification: simulation and formal method-based approaches*. Prentice Hall Professional Technical Reference, 2005.

[21] Te-Chang Lee and Pao-Ann Hsiung. Mutation coverage estimation for model checking. In Farn Wang, editor, *Automated Technology for Verification and Analysis*, pages 354–368, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[22] Andrew Piziali. *Functional verification coverage measurement and analysis*. Kluwer, 2004.

[23] Warren S Sarle. Measurement theory: Frequently asked questions. *Disseminations of the International Statistical Applications Institute*, 1(4):61–66, 1995.

[24] Shuo Yang, Robert Wille, and Rolf Drechsler. Improving coverage of simulation-based verification by dedicated stimuli generation. In *2014 17th Euromicro Conference on Digital System Design*, pages 599–606, 2014.