Quantifying Software Reliability via Model-Counting*

Samuel Teuber[0000−0001−7945−9110]** and Alexander Weigl[0000−0001−8446−4598]

samuel@samweb.org, weigl@kit.edu
Karlsruhe Institute of Technology (KIT)
Karslruhe, Germany

Abstract. Critical software should be verified. But how to handle the situation when a proof for the functional correctness could not be established? In this case, an assessment of the software is required to estimate the risk of using the software.
In this paper, we contribute to the assessment of critical software with a formal approach to measure the reliability of the software against its functional specification. We support bounded C-programs precisely where the functional specification is given as assumptions and assertions within the source code. We count and categorize the various program runs to compute the reliability as the ratio of failing program runs (violating an assertion) to all terminating runs. Our approach consists of a preparing program translation, the reduction of C-program into SAT instances via software-bounded model-checker (CBMC), and precise or approximate model-counting providing a reliable assessment. We evaluate our prototype implementation on over 24 examples with different model-counters. We show the feasibility of our pipeline and benefits against competitors.

Keywords: software verification, software reliability, model counting

1 Introduction

Formal verified safety, defined as the absence of catastrophic consequences [1], yields a high guarantee on the well-functioning of critical software. But proving safety is a hard and tedious process due to the necessary formalization, verification and (possibly) bug fixing for a given software system. In cases where proof cannot be established, other techniques for the reliability assessment of the software are required. To this end, we want to quantitatively estimate the risk of usage of an assessed software. Traditionally, safety is a qualitative property that a software might or might not fulfill, whereas reliability is often a quantitative,

* The final authenticated version is available online at https://doi.org/10.1007/978-3-030-85172-9_4

** This work was supported by funding of the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL)

measurable property, e.g., the likelihood of failure or the failure rate. Quantitative analysis is, however, also a valuable addition to formal safety verification. For example, quantitative analyses are useful to assess the strictness of (input) assumptions imposed for proving software properties. Those are just some reasons why quantitative and probabilistic software analysis has been identified as an interesting research topic (e.g., [16,23]). In particular, [13,8,7] presented approaches to combine quantitative analysis with symbolic execution.

Contribution. In this work we present a formal approach for the quantification of the violation or adherence to a functional specification through exact and approximate model counting. The approach processes C-programs, where the program specification is given via assertions and assumptions in the procedure bodies. The first step in the approach is a behavior-preserving program translation, which makes the violation or adherence of specification in program runs countable. Afterwards, we use CBMC [5] to convert the transformed C-program into multiple CNF formulae. By using CBMC, our approach is limited to C-programs with a bounded execution and bounded data domain. However, by leveraging CBMC's bit precise semantic, we have wider and more precise support of C-programming language in comparison to previous work. Each model of the CNF formulae represents a possible program run, which we count precisely or approximately with tools like GANAK [17] or ApproxMC [3,18,15]. We present a sophisticated evaluation of our prototype in comparison to its competitor [6] which shows advantages of the bit-precise support and non-determinism. The prototype is publicly available.[1]

Example. Consider the example in Fig. 1 which computes the square root of a non-negative integer in the upper half of variable `a`. The algorithm was later modified to return the actual integer square root `lower`. Note, this modification is flawed which will be discussed in more detail below. There are multiple aspects of this program which can be quantified. Firstly, we can compute the number of inputs `x` for which above mentioned flaw leads to an assertion miss (violation of the assertion condition) before the return-statement. Secondly, we can compute the number of inputs `x` for which the assume-statement fails. While the number of assertion misses is a measure stating to what degree the program at hand is flawed, the number of assumption misses describes how tight the

```
#define TOP2BITS(x) ((x & (3 << 30)) >>
    30)
int sqrt(int x) {
  assume(x>=0);
  int input = x;
  int a = 0, r = 0, e = 0;
  for (int i = 0; i < 32; i++) {
    r = (r << 2) + TOP2BITS(x);
    x <<= 2; a <<= 1;
    e = (a << 1) + 1;
    if (r >= e) { r -= e; a++; }
  }
  int lower = (a>>16);
  unsigned int upper = lower+1;
  assert(lower*lower<=input
    && upper*upper>input);
  return lower;
}
```

Fig. 1: Variation of Square Root Algorithm in [20]

[1] https://github.com/samysweb/counterSharp

assumptions are under which we try to guarantee correct behavior. In Fig. 1, for example, our assumptions exclude half of the possible input space (namely any negative value). Depending on the use case, this might be considered a very strong assumption which might not match reality.

Overview The formal foundations of our approach, including the definition of reliability, is in Sec. 2. In Fig. 4, we begin building the pipeline, which contains the program transformation, model-counting, and the correctness (Sec. 3). The comparison with [6] and runtime statistics is given in the evaluation (Sec. 4).

2 Foundations of the Pipeline

```
int test1(int x) {
  assert(x!=0);
  return v/x; }
int test2(int x) {
  x = (x<0)? 0 : x;
  int *y = malloc(sizeof(int));
  int div = (*y)*x
  assert(div!=0);
  return v/div;
}
```

Fig. 2: In `test1` the failure only depends on the input. Differently, in `test2` the failure depends on input and nondeterminism.

Traditionally, reliability of a program is quantified via the number of inputs for which there exists a program path leading to failure in comparison to the number of inputs for which no such program path exists (as seen for example in [7]). However, this definition draws an incomplete image of the software's reliability for nondeterministic programs.

For example, consider the code in Fig. 2: Both functions are prone to a divison by zero failure which is caught by an assertion. The functions differ, because only in `test1` this failure is solely dependent on the input where the division by zero failure *must* happen for exactly one input value (specifically for `x` set to `0`). Meanwhile, the division by zero failure *may* happen for any input of the function `test2` in Fig. 2 while it *must* happen for a negative input value for `x`. We use this semantic of *must* and *may* failures in the following analysis. A failure *must* occur for an input if an assertion miss occurs for every program path. A failure *may* occur for an input if there is both, a program path leading to an assertion miss and a program path leading to an assertion hit.

Partitions of the input space. Formally, we consider a program as a relation $P \subseteq \mathcal{S}^2$ between start state $s \in \mathcal{S}$ and the final state $s' \in \mathcal{S}$. We denote this relationship with $s \xrightarrow{P} s'$. We distinguish between the input values $i \in \mathcal{I}$ and output $o \in \mathcal{O}$ of a program: The input values $i = s\lfloor_{\mathcal{I}}$ are part of the start state, whereas the output values $o = s'\lfloor_{\mathcal{O}}$ are part of final state. $s\lfloor_{\Sigma}$ denotes the projection to the variables given from the set Σ. This formalization allows choosing arbitrary

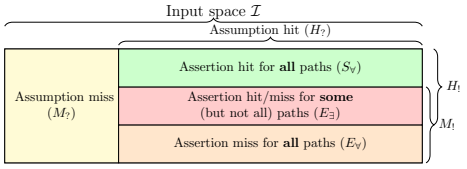

Fig. 3: Partitions $M_?$, $H_?$, $E_\forall$, $E_\exists$, and $S_\forall$ of a programs input space $\mathcal{I}$

values of the non-input variables (e.g., global or local variables) in the start state of a program. Thus, for a given input i, there might be multiple possible start states. And given a single start state s, there might be a set of reachable final states s'.

In combination, given an input value $i \in \mathcal{I}$, the reachable output values are denoted by $\mathfrak{O}(i) = \{o \mid \ s\lfloor_{\mathcal{I}} = i \ \wedge \ s \xrightarrow{P} s' \ \wedge \ o = s'\lfloor_{\mathcal{O}}\} \subseteq \mathcal{O}$. Moreover, we introduce the function $check\colon \mathcal{S} \to \{✓, 🐞, ⊘\}$, which can determine whether the program executed normally (✓, adhering all reached assumptions and assertions), or abnormally terminated denoting the first violation of an assertion (🐞) or an assumption (⊘) given the final state s'. We lift *check* to a set of states, i.e., $check(\mathfrak{O}(i)) \subseteq \{✓, 🐞, ⊘\}$ denotes with the possible outcomes for a given input. Later, in Sec. 3.1 we weave the functionality of *check* into the given C-program.

With the definition of *check* we can partition the input space $\mathcal{I}$ of a program into following parts (Fig. 3) under an additional assumption: the given assume-statements within program are only referring to the input variables of the program. The partitions are defined as follows:

$$M_? = \{i \mid check(\mathfrak{O}(i)) = \{⊘\}\} \tag{1}$$
$$H_? = \{i \mid ⊘ \notin check(\mathfrak{O}(i))\} \tag{2}$$
$$E_\forall = \{i \mid check(\mathfrak{O}(i)) = \{🐞\}\} \tag{3}$$
$$E_\exists = \{i \mid check(\mathfrak{O}(i)) = \{✓, 🐞\}\} \tag{4}$$
$$S_\forall = \{i \mid check(\mathfrak{O}(i)) = \{✓\}\} \tag{5}$$

In terms of notation, we denote assumption related variables with ? and assertion related variables with !. M represents *misses* and H represents *hits*, whereas E represents *error* and S represents *success*. Finally, r is used for ratios. Thus, $M_?$ describes the partition of invalid input values according to our assumptions. There can not occur any other observations for these inputs. $H_?$ on the other hand represents all input values which correspond to our assumptions. This can further be split into $E_\forall, E_\exists$ and $S_\forall$. $E_\forall$ represents the input values that always lead to an error, where $E_\exists$ are the input values, where sometimes an error occurred. $S_\forall$ are the input values that always adhere to the assumptions and assertions in the program, regardless of the value of the non-input variables in the start state.

Besides $M_?$ and $H_?$ the model counting approach enables us to measure the following to metrics: $H_! = S_\forall + E_\exists$ and $M_! = E_\forall + E_\exists$

Through suitable subtraction using $H_?$ we can then compute $E_\forall, E_\exists$ and $S_\forall$. It remains a design choice which input partitions of Fig. 3 are treated as error or success. Note, for determinstic programs the partition $E_\exists$ is empty, leading to simpler calculation and less model-counting calls. Depending on the use case we might then be interested in ratios describing how many of the inputs show a particular behavior. To this end, we define the following, exemplary, ratios:

$$r^m_? = \frac{M_?}{\mathcal{I}} \qquad r^f_\forall = \frac{S_\forall}{H_?} \qquad r^e_\forall = \frac{E_\forall}{H_?} \tag{6}$$

3 Pipeline

Fig. 4 shows the three pipeline steps of our approach. Firstly, we weave the observation of the *check* function into the given original C-program, by transforming the program flow in such a way that a violation of an assumption or an assertion are explicitly stored in the program state (Sec. 3.1). Also, a violation of either an assumption or an assertion leads to an early termination. The C-program input consists of an entry routine and the (transitively) required routines. Secondly, we use CBMC, a bounded model-checker for C-programs, for the transformation of the program into multiple CNF formulae (Sec. 3.2) where we have one CNF formula for each measured quantity. Thirdly, we run model-counting tools, like ApproxMC or GANAK, on the CNF formulae and thus obtain required metrics (Sec. 3.3) to calculate the reliability.

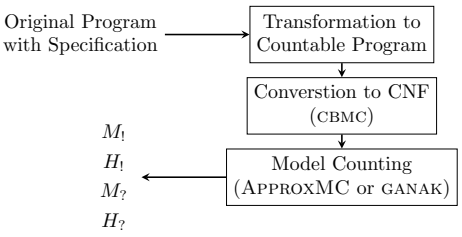

Fig. 4: Overview on the quantification pipeline for the computation of hit and misses of assumption and assertion within the original program.

3.1 Transformation: Make Violation Countable

The goal of the program transformation is to make the assumption or assertion violations explicit in the program state by using dedicated fresh variables. Therefore, the violations become countable by Model Counters. An example of the transformation is presented in Fig. 5. Note, we use the constants `true` and `false` for convenience reason, although they are not defined in standardized C.

Restrictions on C-programs. The allowed C-subset is restricted by CBMC, which allows both basic datatypes (int, float, char etc.) and complex datatypes (e.g., arrays and structures). Subroutine calls are limited to stand-alone calls and calls assigned to a variable (i.e., no nested subroutine calls and no calls directly within a return-statement). Such calls can easily be transformed into non-nested, non-return calls by introducing appropriate local variables – even automatically if desired. While complex datatypes are supported, they cannot be used as input variables for model counting directly, but must be constructed within the function under evaluation explicitly (e.g., by passing an array's elements into the function explicitly and then constructing the array within the function). We assume that assume-statements solely express conditions imposed on the program's input (and not program internal behavior) and are thus positioned at the beginning of the program before any assertion statements.

Transformation. In the example in Fig. 5, the transformation is applied to the entry function `test`. Required sub-routines are considered for transformation accordingly.

```
1
2  int subroutine(int y) {
3
4    assert(y<0);
5    return -y;
6
7
8  }
9
10 int test(int x, int y)
11 {
12
13   assume(x>0);
14   int z = 0;
15   if (y < 0) {
16     z += subroutine(y);
17
18     return z;
19
20
21
22
23
24
25   }
26   assert(z>=0);
27   return z+x;
28
29 }
```

```
1  char am = false; char as = false;
2  int subroutine(int y) {
3    int rv;
4    if (!(y < 0)) { as = true; goto end; }
5    rv = -y;
6  end:
7    return rv;
8  }
9
10 int test(int x, int y)
11 {
12   int rv;
13   if (!(x > 0)) { am = true; goto end; }
14   int z = 0;
15   if (y < 0) {
16     z += subroutine(y);
17     if (as || am) goto end;
18     rv = z;
19 end:
20     assert(am || as); //assertion hit
21     assert(!as);//assertion miss
22     assert(am); //assumption hit
23     assert(!am);//assumption miss
24     return rv;
25   }
26   if (!(z >= 0)) { as = true; goto end; }
27   rv = z + x;
28   goto end;
29 }
```

Fig. 5: Left is the original program, and right the program transformation as described in Sec. 3.1

The first step introduces two global variables initialized by `false` (`am` and `as`) which store whether an assertion or assumption was missed in global program state (Line 1). This implies that assume- and assert-statements actually have to modify this variable after checking their specific criterion. Therefore, such statements are transformed into an if-statement with their negated original condition. Missing a condition sets the corresponding variable (`am`/`as`) and initiates a jump to the end of the current function. This transformation can be observed in Lines 13 and 26. Note that the same transformation also takes place in subroutines as can be seen in Line 4. In order to support sub-routine calls, every such call triggers a check for assertion- or assumption misses afterwards. If a sub-routine call has violated an assertion, the callee routine directly jumps to its end. This behavior occurs recursively leading to an early termination of the program – similar to exception handling in modern programming languages.

We then unify all return-statements of a routine into a single, labeled return (e.g., Line 18) targeted by multiple goto-statements which replace the return statements in the old program and direct the program flow to the one remaining return statement (e.g., Line 27). Line 18 shows the first return-statement of a routine is kept and labeled with `end`, while all other return-statements (e.g., Line 27) are replaced by goto-statements. A new value variable `rv` is introduced which stores the return value across this jump (Lines 12 and 3). If no return-

statement exists within a subroutine, we introduce a labeled dummy return-statement at the end of the routine.

Finally, the four assert-statements in Lines 20 to 23 (with conditions as explained later in Sec. 3.2) are added just before the return-statement of the entry routine. Only one of these assert-statements is inserted for the conversion to the CNF formulae in order to generate specific formulae for counting the assumption/assertion hits and misses.

In the remainder of this paper, if we talk about assertion, we mean the assertion given as the specification in the original program, whereas the notion of an assert-statement refers to (one of) the four assertions in the transformed program (Lines 20 to 23).

3.2 Conversion into CNF

For the conversions of C-programs into CNF formulae we use CBMC. CBMC converts the transformed program p and a specification ψ into a CNF formula $\phi = \pi \wedge \neg\psi$ where π is a CNF representation of the unrolled program (w.r.t. a certain loop iteration or recursion depth). Formula ϕ is then satisfiable iff there exists a program path in p leading to a violation of the given specification ψ. Using CBMC with its builtin bit precise semantic, we generate four formulae with varying specification (assert-statements) before the final return-statement: `am`, `!am`, `am || as` and `!as`. Additionally we generate a fifth formula which allows to compute the number of inputs for which the bounds of CBMC were insufficient. The last assert-statement, for instance, is transformed into a formula asserting that `as` is `true` and can thus be used to count the number of models which produce an assertion miss (and thus to compute $M_!$). Note, that enabling only the required assert-statement allows CBMC to reduce the size the generated CNF formula by slicing which improves performance of the model-counters.

3.3 Model Counting in the Pipeline

Preliminaries. For our approach we make use of model counting under projection. Given a CNF formula ϕ over the signature Σ, $|\,\mathrm{models}(\phi\lfloor_\Delta)|$ denotes the number of satisfying assignments (models) of ϕ projected on the variables $\Delta \subseteq \Sigma$, where the formula $\phi\lfloor_\Delta$ denotes the projection of ϕ on Δ. Therefore, $\phi\lfloor_\Delta$ is the strongest formula over Δ which is entailed by ϕ [12, Logical Foundations] and states the same constraints on the atoms in Δ as ϕ. We define $\mathrm{models}_\Delta(\phi) = \mathrm{models}(\phi\lfloor_\Delta)$. For our use case, we distinguish between exact and approximative model-counters. Exact (e.g., sharpSAT [22]) or probabilistic exact (e.g., GANAK [17]) model counters compute the exact number of models t with a certain probability $\delta \leq 1$. Additionally, approximative or (δ, ϵ) model counters (e.g., ApproxMC [3,12,18,15]), return an estimated count $\tilde{c}$ with the guarantee of a relative error ϵ and a maximum uncertainty of δ: $Pr(\tilde{c} \in [t/(1+\epsilon), (1-\epsilon)t]) \geq 1 - \delta$

The parameters ϵ and δ are given by the user. We further elaborate on model counting in Appendix A.

Measuring the Reliability. In Sec. 3.2 we obtain formulae which are satisfiable iff there is a start state leading the encoded program to violate the encoded specification. As noted earlier, we encode the inverse of the specification we are interested in and can thus quantify the number of inputs adhering to the specification at hand by using projected model counting with Δ containing exactly the propositional variables corresponding to the program input (a more detailed argument for the correctness of this approach is available in Sec. 3.4). Given the measured counts $\psi_?^m, \psi_?^h, \psi_!^m, \psi_!^h$ (respectively assumption miss and hit, assertion miss and hit), the computation for the case of exact results is relatively straight forward:

$$\begin{aligned} M_? &= \psi_?^m & E_\forall &= \psi_?^h - \psi_!^h \\ H_? &= \psi_?^h & E_\exists &= \psi_?^h - E_\forall - S_\forall = \\ S_\forall &= \psi_?^h - \psi_!^m & &= \psi_!^h + \psi_!^m - \psi_?^h \end{aligned}$$

For the case of approximate model counting using ApproxMC, each of the given counts are burdened with an uncertainty (δ, ϵ). Note, that these uncertainties are further propagated when we compute our ratios in (6). As the ratio $r_?^m$ only depends of the count $M_?$, its error (δ, ϵ) is just propagated towards $r_?^m$. For the ratios $r_\forall^e$, and $r_\forall^f$ the (δ, ϵ) error of each approximated count are multiplied, i.e., the error bound is $(1+\epsilon)^2$ with certainty of $(1-\delta)^2$. We clearly see that the ratios' become less precise.

We consider two numerical examples (for details see Sec. 4) to explore the error bounds[2]. Let us assume ApproxMC's standard parameters $\delta = 0.2$ and $\epsilon = 0.8$. First, consider the case of rangesum03. For this benchmark the following values represent the correct model counts: $H_? = 2^{96}$, $M_! = 2^{64}$, $H_! = 2^{96} - 2^{64}$. By computing the bounds (see Appendix A) we obtain the following error bounds with a probability of 0.64 each:

$$\text{For } r_\forall^e\text{: } -2.23 \leq \frac{\psi_?^h - \psi_!^h}{\psi_?^h} \leq 0.69 \quad \text{For } r_\forall^f\text{: } \quad 0.99\ldots \leq \frac{\psi_?^h - \psi_!^m}{\psi_?^h} \leq 0.99\ldots$$

We see a strong asymmetry between the error bounds for $r_\forall^e$ and those for $r_\forall^f$ caused by the strong asymmetry of $\psi_!^h$ and $\psi_!^m$. Note that for deterministic benchmarks the ratios can be computed through one minus the other ratio respectively. As a second case, consider the benchmark usqrt-broken for which the correct values are given by: $H_? = 2^{32}$, $M_! = 2^{31}$ and $H_! = 2^{31}$. Here, the split between assertion misses and assertion hits is essentially even. We obtain the following (equal) error bounds both for $r_\forall^e$ and $r_\forall^f$: $-0.62 \leq \frac{\psi_?^h - \phi}{\psi_?^h} \leq 0.89$ for $\phi \in \{\psi_!^h, \psi_!^m\}$.

Note that, firstly, for cases where $\psi_!^m$ and $\psi_!^h$ are expected or found to be of similar magnitude, a stricter value for ϵ should be considered. Secondly, in all cases it is worthwhile to examine, both, the ratios and all the absolute numbers:

[2] For conciseness, we mostly round the given bounds to two decimal places.

While a success ratio $r_{\forall}^{f} \geq 0.99$ seems good, it might still be the case that a large number of inputs yield an assertion miss if the input space $\mathcal{I}$ and $H_?$ are sufficiently large (as seen for case rangesum03 in Sec. 4).

3.4 Correctness of the pipeline

The pipeline we described in Sec. 3 is correct. By correctness, we mean that we get correct counts $\psi_?^m, \psi_?^h, \psi_!^m$ and $\psi_!^h$ for $M_?, H_?, M_!$ and $H_!$ when running the pipeline with an exact model counter. The correctness depends on three elementary properties: First, the program transformation preserves the behavior of the original program. Of course, a new program flow is established, but in cases without a violation of an assumption or assertion the transformed program behaves equally. Secondly, every hit or miss of an assumption or assertion is captured faithfully by a violated assert-statement (Lines 20 to 23). Thirdly, every (projected) model of the generated CNF formulae is indeed a representative for a violating or valid program path. In the Appendix B we elaborate the correctness deeply.

4 Evaluation

In order to evaluate our approach, we ran our prototypically-implemented pipeline on a number of C-programs from various benchmark families with the objective of showing its strengths, weaknesses and limits. Our experiments aimed to answer the following questions:

(Q1) Does our pipeline admit the quantification of more complex programs in comparison to [6]?
(Q2) How large can programs and input spaces become for given resource limits?
(Q3) Where does our bit-precise semantic help in obtaining more precise results and where is it too costly in comparison to [6]?
(Q4) Can our pipeline accurately quantify non-deterministic program behavior?

Implementation. For our experiments we implemented CounterSharp[3] as a tool which transforms input C-programs into countable CNF formulae using CBMC. We apply the transformation (Sec. 3.1) on the input C-program's abstract syntax tree. Afterwards, the modified abstract syntax tree is converted back into C-code which is automatically passed to CBMC producing a total of five CNF formulae: Four checking for assertion/assumption hits and misses and another formulae to check for how many inputs (if any) the given bound is insufficient (i.e., for which inputs there are paths which need a deeper unroll of loops). The formulae provided by CBMC are adjusted to contain model counting projection instructions. We extract the propositional variables which represent the inputs variables in the C-program. Finally, our tool returns the five formulae containing

[3] CounterSharp counts (thus sharp) counterexamples (thus counter) for a given specification.

projection instructions, which can then be processed by a suitable model counter of the user's choice. This approach both leaves the freedom to make use of other model counters and allows the trivial parallelization of this last quantification step. Our tool is publicly available[4].

Experiment Setup. All experiments were run on a 4 core Intel Core i5-6500 processor and 16 GB of RAM. CounterSharp was run with a timeout of 15 min. while the model counters had 5 minutes per instance. In order to achieve a fair comparison, the tool by [6] was given a timeout of 40 minutes to account for the multiple model counter runs in the case of CounterSharp. All runs had a restricted memory of 2 GB. The execution of the tools was monitored using the *runlim* utility [2]. All scripts and raw results are available online [21]. The stated performance data is given as the median of 5 runs. ApproxMC was used in default configuration (though we used activated sparse hashing) and varying seeds across the runs, analogue for GANAK.

In the remaining, we compare varying setups: The setup `dim` denotes the analyzer of [6]. The setup `cS-gan` consists of CounterSharp with GANAK for model counting for determinstic programs. The runtime is given as the time of CounterSharp c added by the minimum runtime of (parallel) counting assertion hits $g_{H_!}$ and misses $g_{M_!}$: $c + \min(g_{H_!}, g_{M_!})$. Analogue, CounterSharp with ApproxMC (`cS-app`), where the runtime is given as $c + \max(a_{H_!}, a_{M_!})$. Both setups `cS-gan` and `cS-app` simulate a run of CounterSharp followed by a parallel run of GANAK or ApproxMC to compute assertion hit/miss counts. Additionally, we defined two setups used for non-deterministic cases named `cSn-gan` and `cSn-app`, where the runtime is given as $c + \max(\min(g_{H_?}, g_{M_?}), g_{H_!}, g_{M_!})$ and $c + \max(a_{H_?}, a_{M_?}, a_{H_!}, a_{M_!})$. Here, including extra counts to obtain reliable results for assumption hits, assertion hits and assertion misses—a necessity because of the possible overlap between assertion misses and hits in the case of non-determinism. GANAK can stop this computation once one of the two counters for assumption hits and misses returns, as the complement can be computed through subtraction. On the other hand, ApproxMC has to count both due to approximation errors (hence the max).

Benchmarks by [6]. In a first step we compared `cS-gan` and `cs-app` directly with `dim` using the benchmark set presented in their original work where each benchmark provided a version with $\mathcal{I}$ of size 10 and 1000. The comparison in the first part of Table 1 clearly shows that neither `cS-gan` nor `cS-app` can win against `dim` on the paper's original benchmark set. Indeed, it seems `dim` is very well suited to answer quantification questions on their benchmark set. To illustrate the reasons for its success, it is worth having a look at the considered

```
int testfun(int n) {
  assume(n>=0&&n<=999);
  int x=n, y=0;
  while(x>0) { x--; y++;
      }
  assert(y==n);
}
```

Fig. 6: Benchmark count_up_down for input size 1000

[4] https://github.com/samysweb/counterSharp

Table 1: Evaluation of `dim` [6], `cS-gan`, `cS-app`, `cSn-gan`, `cSn-app` on the applicable benchmarks: Runtimes in seconds (median of 5 runs). (MO = out-of-memory, TO = timeout)

Deterministic Benchmarks

Comparison to `dim`[6]													
Benchmark	Source	`dim`				`cS-gan`				`cS-app`			
Input Size		10		1000		10		1000		10		1000	
		time	exact	time	exact	time	exact	time	exact	time	exact	time	exact
bwd_loop1a	[6]	**0.51**	✓	**0.55**	✓	1.68	✓	2.82	✓	4.98	✓	8.68	≈
bwd_loop2	[6]	**0.67**	✓	**0.41**	≈	1.76	✓	1.35	✓	2.67	✓	4.24	≈
count_up_down	[6]	**0.54**	✓	**0.27**	✓	1.16	✓	8.75	✓	1.19	✓	142.45	≈
example1a	[6]	**0.64**	✓	**0.46**	✓	1.59	✓	1.83	✓	5.18	✓	10.02	≈
example7a	[6]	**0.54**	✓	**0.82**	✓	1.07	✓	TO	—	1.76	✓	TO	—
gsv2008	[6]	**0.43**	✓	**0.62**	✓	1.6	✓	8.89	✓	3.03	✓	79.41	≈
hhk2008	[6]	**0.44**	✓	**0.68**	✓	0.96	✓	TO	—	1.56	✓	TO	—
Log	[6]	**0.57**	≈	**0.69**	≈	1.45	✓	TO	—	1.90	✓	TO	—
Mono3_1	[6]	**0.48**	≈	**0.74**	≈	TO	—	TO	—	TO	—	TO	—
Waldkirch	[6]	**0.4**	✓	**0.37**	✓	1.35	✓	8.17	✓	1.56	✓	146.56	≈

Complex benchmarks

Benchmark	Source	`dim`	`cS-gan`		`cS-app`	
		reason for failure	time	exact	time	exact
floor-broken	[20]	float	TO	—	**11.13**	≈
floor	[20]	float	**1.07**	✓	8.47	✓
overflow	crafted	incorrect (overflow)	**1.41**	✓	1.69	≈
Problem10_16	[19]/[10]	timeout	TO	—	**723.73**	≈
Problem13_4	[19]/[10]	timeout	TO	—	TO	—
rangesum03	[19]/[4]	arrays	**0.51**	✓	2.0	≈
rangesum05	[19]/[4]	arrays	TO	—	TO	—
usqrt-broken	[20]	bit arithmetic	TO	—	**17.06**	✓
usqrt	[20]	bit arithmetic	TO	—	**105.83**	✓

Nondeterministic Benchmarks[5]

Comparison to `dim`[6]				
Benchmark	Source	`dim`	`cSn-gan`	`cSn-app`
		time	time	time
bwd_loop10-2	[6]	**0.39**	MO	MO
bwd_loop10	[6]	**0.65**	MO	MO
bwd_loop7-2	[6]	**0.55**	230.11	6.41
bwd_loop7	[6]	**0.88**	6.36	6.41
example7b-2	[6]	**0.45**	TO	TO
example7b	[6]	**0.17**	4.18	3.63
for_bounded-2	[6]	**0.25**	MO	135.51
for_bounded-1	[6]	**0.67**	2.91	2.17
nondet	crafted	**0.87**	TO	1.83

benchmarks. One of the benchmarks where `cS-app` is particularly bad in comparison to `dim` is the count_up_down benchmark in Fig. 6. It is clear that a bounded model checker approach is worse at solving instances like this one due to the large number of loop unrolls necessary. An abstract interpretation approach like `dim`, seems to handle this kind of task a lot better—and independent of the loop size. This difference becomes even clearer for the case of the benchmark Mono3_1 which contains a loop repeated one million times. As a matter of fact, every time `cS-gan` or `cS-app` take more than 15 seconds or time out on the benchmark set in the first part of Table 1, it is due to a benchmark which requires an unrolling of a loop with depth larger or equal to 500.

The only exception to this rule is the benchmark gsv2008 which takes longer than 15 seconds for a run with unroll depth 101. Conversely, nearly all benchmarks executed within 15 seconds require a less deep loop-unrolling. It thus

seems that loop depth is the main bottleneck for our approach on this benchmark set. At the same time, the program logic of all benchmarks in the set is comparatively simple: subtraction and addition paired with while, for or if-statements. Towards answering (Q3) we observe that the approach proposed by Dimovski and Legay [6] works very well on this type of benchmark. However, the question remains whether more complex benchmarks might require a more precise semantic than the one available in `dim`.

Complex programs. To this end, we looked at a number of other benchmarks from the SV-Competition [19] as well as the C-Snippets [20] code repository. We collected a number of benchmarks which allowed some form of quantification through input variables and either had a wider range of interesting, representative program constructs (such as arrays, floats, or bit arithmetic) or represented suitable candidates to test the scalability of our approach. These benchmarks were manually modified to allow quantification on them (addition of suitable assertion- and assumption-statements) and to compare different sizes (e.g., comparison of 3 element and 5 element array input). Additionally, we crafted one benchmark showing behavior we were particularly interested in. All of the instances mentioned in the second part of Table 1 show behavior which cannot be analyzed by `dim`: The tool was either unable to analyze the programs due to the use of arrays and bit arithmetic, ran into a timeout due to instance size or even produced wrong results due to the neglection of overflows. We discuss these benchmarks as well as why we believe they are difficult and what our results show.

The first two benchmarks in the table represent a broken and correct algorithm to compute the floor function of a floating point number. For the experiment we assumed a positive, non-infinite, non-NaN 128 bit `long double` input which the program is supposed to round down into another `long double` number using bit arithmetic. The final assertion checks whether the computed number is smaller than the original number. We added a broken version of the program which, according to `cS-app`, breaks the specification for approx. $1.5{*}2^{95}$ of the 2^{127} positive inputs. The `cS-app` tool also returns that 2^{127} inputs comply to the specification. Here we see the reason why it is necessary to compute both counts (assertion miss and assertion hit): The ratio between assertion hits and misses is approx. $1.75 * 10^{-10}$ and thus the entirety of assertion misses well lies within the error bounds of the assertion hit measurement. Therefore, it seems good advice to always inspect the numbers and their relations manually in order to spot such approximation errors during data interpretation.

The float instances are followed by the only instance which `dim` was able to compute faster than our tools and which is a crafted benchmark. However, `dim` returns faulty results due to an integer overflow which remains undetected in polyhedra: The benchmarks assumes an input `x` strictly larger than `INT_MIN/2`. If `x` is negative, `|INT_MIN/2|` is subtracted while `|INT_MIN/2|` is added if `x` is positive. The final assertion requires that `x` be negative. `dim` returns that the assertion is met by one third of the inputs (i.e., by all inputs which are negative) and this is of course what a polyhedra tool would have to return due to

its abstract domain. However, closer examination shows that the addition in the positive case leads to an overflow for any number larger than `|INT_MIN/2|`. Thus, the assertion is actually hit for two thirds of all inputs (i.e., all negative inputs and all sufficiently large positive inputs). This benchmark was of course crafted to show the drawbacks in the use of polyhedra for quantification. While one might argue that this is an unfair comparison, we merely want to draw attention to the fact that such mistakes are not uncommon in programming and can be better quantified with a tool with bit-precise semantic, like counterSharp.

The following two benchmarks stem from the RERS Challenge 2018 [10] and concern the reachability problem for linear time logic (LTL) formulae. For each instance a number of error states are defined which should not be reached. The states reached by the program depend on the input variables. `Prob10` is then run for 16 time steps converting the program in a way that it takes 16 char variables as inputs, while `Prob13` was only supposed to be run for 4 time steps. As only values from 1 to 5 (instance 10) or 1 to 10 (instance 13) are allowed, suitable input restrictions were introduced for both `dim` and `cS-⋆`. The original RERS challenge had 3 problem levels of which `Prob10` was level *small* and `Prob13` was level *medium*. `Prob10` consists of approx. 1.3k LoC while `Prob13` contains around 114k LoC. While `cS-app` works relatively well on the small instance, counterSharp runs into a timeout for the medium size instance when trying to construct the CNF formulae. On the other hand, `cS-gan` cannot compute either of the two benchmarks. This is probably due to the fact that the number of clauses (and variables) in the instance are at least one order of magnitude larger than the numbers in other benchmarks solvable by `cS-gan`. It turned out that `dim` seems to be unable to solve both instances due to a memout. We believe the reason for this is the scale of the benchmark instances which is much larger than instances previously considered in the evaluation of `dim`.[6]

The following two rangesum examples implement the computation of a rangesum for an array of size 3 and 5. This benchmark can, again, only be evaluated for the `cS-⋆` tools as it requires the ability to handle arrays. We modified the benchmark in such a way that the elements inside the array are given as input parameters to the main function thus allowing an easy quantification across all possible array values. Both, `cS-app` and `cS-gan` are able to quickly solve the 3 element instance. We also see that `cS-gan` is particularly efficient in cases where one of the two counts (here the assertion hits) can be computed particularly fast, as the complementary count (which in this case would have timed out) can be computed exactly. At the same time, both tools still fail to solve the 5 element instance. This can be explained by the vastly larger input space for an array of 5 integer elements which considers an input space 2^{64} times larger than the input space for 3 elements.

The final two benchmarks stem from Fig. 1 which represents the computation of an integer square root. As we already mentioned in the introduction, the code

[6] Given the choice of a relatively low memout initially, we reran `dim` on Problem_10 with 8 GB of memory available. However, the program ran into a timeout after 2400s using 4.5 GB of memory.

in the listing is flawed (this corresponds to usqrt-broken). Namely, the shift in Line 12 is a signed right shift which introduced flawed results for any input `x` larger than 2^{30}. Indeed, `cS-app` correctly returns in all 5 runs that the assertion succeeds (resp. fails) for 2^{30} (resp. also 2^{30}) inputs while 2^{31} inputs already miss the assumption. For the case of the fixed version (usqrt) the main challenge for `cS-app` (or its underlying SAT solver to be precise) is showing the unsatisfiability for of the assertion miss formula which takes $104.93s$ in comparison to $3.01s$ for counting all 2^{31} assertion hits.

Concerning questions (Q1) and (Q2) we see that there is a category of benchmarks which the tool by Dimovski and Legay [6] fails to analyze and in all fairness was probably never meant to analyze given the polyhedral domain approach. On the other hand our tools `cS-gan` and `cS-app` are able to solve a number of these (in our opinion) interesting benchmarks. In particular our tools admit the quantification of larger programs (e.g., Problem_10) and more complex programs (i.e., arrays or advanced arithmetic). Concerning (Q3) we showed how a bit precise semantic helps in correctly quantifying overflow errors.

Non-deterministic programs. Turning to the question of non-deterministic programs (Q4), we can observe in the last part of Table 1 that all benchmarks considered are solved the fastest by `dim`. We observe, once again, that the cases where `cSn-gan` or `cSn-app` yield a timeout or memout are cases with very deep loops with the notable exception of the benchmark nondet which does not involve loops. While `dim` is the fastest tool, it turns out that `cSn-gan` and `cSn-app` can sometimes produce more precise results than the approximate result yielded by `dim`. In particular, the benchmark nondet which corresponds to the code in `test2` in Fig. 2 demonstrates this behavior. As previously discussed, the assertion can fail for any input due to the non-deterministic value of `y`, but it must fail for any negative input `x`. Accordingly, `cSn-app` reports a possible assertion miss for 2^{32} cases and a possible assertion hit for 2^{31} cases. This corresponds to a probability of success between 0 and 50 percent and a probability of violation between 50 and 100 percent for a uniform input distribution. While `dim` is faster in reporting its result, it reports that both success and violation probability lie within the range 0 to 100 percent which is a lot less precise than `cSn-app`. The reason for this behavior is quite likely the multiplication in the program which is difficult to handle for the polyhedra abstract domain. Equally, the complexity of the formula due to multiplication might be a reason why the exact counter `cSn-gan` fails to quantify the benchmark. Concerning (Q4) we find that our tools admit the accurate quantification of non-deterministic benchmarks. Just as for deterministic benchmarks our approach is limited by loop depth. Once again certain cases can profit from the bit precise semantic allowing for a more precise quantification (as in the case of nondet).

Analysis of bottlenecks. CBMC and the program transformation take most of the required time for instances with a small input space. As the input space size grows, the counting step either becomes infeasible (especially for GANAK) or requires a lot more computation time in comparison to CBMC (especially for

ApproxMC). Since counting can take significant amounts of time and ganak and ApproxMC showed drastic variations in performance on some benchmarks (e.g. Waldkirch where ganak solves the instance a magnitude faster than ApproxMC or float where the opposite behavior can be observed), an approach using a portfolio of model counters could reduce computation time.

5 Related Work

Klebanov et al. [11] present the idea of using CBMC to generate CNF formulae for model counting to compute the information leakage of a program.

Geldenhuys et al. [8] proposed an approach using symbolic execution for the extraction of path probabilities. To this end, a classic symbolic execution methodology is modified such that it computes probabilities instead of path conditions: At every branching point the algorithm computes the likelihood of descending into the subprocedures at hand under the assumption of a uniform distribution of input variables using the polytope utility LattE [14]. Path probabilities are then computed through multiplication of branch probabilities. Lui and Zhang [13] propose a similar approach which does not use formula slicing or memoization. Filieri et al. [7] applies symbolic execution on Java-programs to extract path conditions for every possible execution path. The paths are then labeled as success, failure or "gray" paths with unknown success status. The number of models for these path conditions are, again, counted with LattE, though they may be constrained by usage profiles describing an input distribution. This allows to compute the probability of failure (failure paths) and the confidence in the result (gray paths) under the given usage profile. The approach furthermore supports multi-threading. In contrast, our approach differs in both: the programming language under evaluation and the counting technique approach allowing more complex behavior than the linear constraints described by polytopes.

The approach of Dimovski and Legay [6] allows the computation of assertion hit and miss probabilities under a uniform input distribution for programs written in a subset of C. The approach uses abstract interpretation to describe the program behavior using the polyhedra domain, which encodes linear constraints between program variables. In a first step, a forward analysis of the program is performed computing an invariant which must hold after the program execution. The forward analysis is based on intervals which have to be defined for the input variables of interest. The forward analysis works by applying sound (but over-approximating) transfer functions on the specified input domain. Afterwards, in a backward pass, conditions specifying an assertion hit/miss are propagated backward using appropriate transfer functions. This approach produces two over-approximating linear constraint preconditions: One for assertion hits and one for assertion misses. The number of inputs corresponding to the precondition at hand are then again computed using LattE [14]. Both results (assertion hit and miss) represent an upper-bound for the number of inputs leading to the specified behavior. By calculating the complement of each count, the tool can further provide a lower bound for each of the two cases. Addi-

tionally, the analyzer provides a semantic that also allows a quantification of non-deterministic programs. While the tool of [6] is similar, it only supports a more restrictive subset of the C programming language in comparison to our approach: The subset supports no complex data types and only integers as basic data types while our work supports both arrays and floats. Additionally, the use of non-linear operators as well as bit operators is greatly restricted. As our tool relies on bounded model checking and SAT model counting, we can avoid this limitation by making use of a bit-precise semantic which also takes overflows into account. On the other hand, our tool has stricter limitations on the loops that can be evaluated as the bounded model checker must unroll such loops up to a sufficient depth for exact results, while Dimovski and Legay [6] can analyze such loops in considerably less time using abstract interpretation if the loop conditions are sufficiently simple.

6 Conclusion

In this work we presented a formal approach allowing the precise quantification of software properties given as assumptions and assertions. The resulting pipeline contains three steps: (1) program transformation, (2) conversion into CNF formulae and (3) model counting. We implemented the pipeline prototypically and undertook an extensive, qualitative evaluation of the prototype. We show that our quantification approach is both, feasible, even a program with 1300 LoC was still analyzable with the given resources, and useful, sometimes yielding results beyond the precision of previous approaches. We further introduced a semantic allowing the quantitative analysis of non-deterministic programs and showed that the pipeline is capable of analyzing such instances, too. In our comparison to the tool in [6], we showed strengths and weaknesses of both approaches: While their approach works very well on programs with high loop depth, our approach allows the quantification of programs using more complex arithmetical operations such as bit operators or multiplication. Equally, we presented a case in which the bit precise semantic helps in returning more precise results for the quantification of non-deterministic programs.

To summarize, we believe that a SAT counting based approach is an interesting addition to the tool set available for software quantification. While some benchmarks also remain out of reach for our approach (either due to the input space explosion with growth in input variables or due to the sheer problem size) and approximation errors need to be considered when analyzing, the capabilities of our approach will scale with advances in model counting in the same way the capabilities of bounded model checkers grow with the advances in SAT solving.

Future Work. The current pipeline can only serve as an estimator of probability of failure for the case of a uniform input distribution. An extension to non-uniform input distributions could further increase the utility of the tool. To this end, we identify a need for *projected* weighted model counters (approximate or exact). Additionally, CBMC allows the use of complex data types such

as structs. However, the current setup only allows basic data types as input variables. A methodology allowing a quantification over a set of complex data structure instances as input might be an interesting addition to the tool in its current form.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. Institute for Systems Research **0114**, 10 (2001). https://doi.org/10.1016/S0005-1098(00)00082-0, http://drum.lib.umd.edu/handle/1903/5952
2. Biere, A.: runlim. Website (2016), http://fmv.jku.at/runlim
3. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In: Kambhampati, S. (ed.) Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016. pp. 3569–3576. IJCAI/AAAI Press (2016), http://www.ijcai.org/Abstract/16/503
4. Chen, Y., Hong, C., Sinha, N., Wang, B.: Commutativity of reducers. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 131–146. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_9, https://doi.org/10.1007/978-3-662-46681-0_9
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **2988**, 168–176 (2004). https://doi.org/10.1007/978-3-540-24730-2_15
6. Dimovski, A.S., Legay, A.: Computing program reliability using forward-backward precondition analysis and model counting. In: Wehrheim, H., Cabot, J. (eds.) Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12076, pp. 182–202. Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_9, https://doi.org/10.1007/978-3-030-45234-6_9
7. Filieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in Symbolic PathFinder. Proceedings - International Conference on Software Engineering pp. 622–631 (2013). https://doi.org/10.1109/ICSE.2013.6606608
8. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic Symbolic Execution. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 166–176. ACM (2012). https://doi.org/10.1145/2338965.2336773
9. Hechter, M., Fichter, J.K.: Model Counting Competition 2020. Website (2020), https://mccompetition.org/, accessed 5th December 2020
10. Jasper, M., Mues, M., Schlüter, M., Steffen, B., Howar, F.: RERS 2018: Ctl, ltl, and reachability. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings,

Part II. Lecture Notes in Computer Science, vol. 11245, pp. 433–447. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_27, https://doi.org/10.1007/978-3-030-03421-4_27
11. Klebanov, V., Manthey, N., Muise, C.J.: SAT-Based Analysis and Quantification of Information Flow in Programs. In: Joshi, K.R., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8054, pp. 177–192. Springer (2013). https://doi.org/10.1007/978-3-642-40196-1_16, https://doi.org/10.1007/978-3-642-40196-1_16
12. Klebanov, V., Weigl, A., Weisbarth, J.: Sound probabilistic #SAT with projection. Electronic Proceedings in Theoretical Computer Science, EPTCS **227**, 15–29 (2016). https://doi.org/10.4204/EPTCS.227.2
13. Liu, S., Zhang, J.: Program analysis: From qualitative analysis to quantitative analysis. Proceedings - International Conference on Software Engineering pp. 956–959 (2011). https://doi.org/10.1145/1985793.1985957
14. Loera, J.A.D., Hemmecke, R., Tauzer, J., Yoshida, R.: Effective lattice point counting in rational convex polytopes. J. Symb. Comput. **38**(4), 1273–1302 (2004). https://doi.org/10.1016/j.jsc.2003.04.003, https://doi.org/10.1016/j.jsc.2003.04.003
15. Meel, K.S., Akshay, S.: Sparse Hashing for Scalable Approximate Model Counting: Theory and Practice. In: Hermanns, H., Zhang, L., Kobayashi, N., Miller, D. (eds.) LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020. pp. 728–741. ACM (2020). https://doi.org/10.1145/3373718.3394809, https://doi.org/10.1145/3373718.3394809
16. Orso, A., Rothermel, G.: Software testing: A research travelogue (2000-2014). Future of Software Engineering, FOSE 2014 - Proceedings pp. 117–132 (2014). https://doi.org/10.1145/2593882.2593885
17. Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: A Scalable Probabilistic Exact Model Counter. In: Kraus, S. (ed.) Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019. pp. 1169–1176. ijcai.org (2019). https://doi.org/10.24963/ijcai.2019/163, https://doi.org/10.24963/ijcai.2019/163
18. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. pp. 1592–1599. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33011592, https://doi.org/10.1609/aaai.v33i01.33011592
19. SoSy-Lab LMU: SV-Benchmarks (2020), https://github.com/sosy-lab/sv-benchmarks
20. Stout, B.: C Snippets (2009), http://web.archive.org/web/20101204075132/http://c.snippets.org/
21. Teuber, S., Weigl, A.: Evaluated artifact for "quantifying software reliability via model-counting" (2021). https://doi.org/10.5445/IR/1000134169
22. Thurley, M.: sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In: Biere, A., Gomes, C.P. (eds.) Theory and Applications of

Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4121, pp. 424–429. Springer (2006). https://doi.org/10.1007/11814948_38, https://doi.org/10.1007/11814948_38

23. Visser, W., Bjørner, N., Shankar, N.: Software engineering and automated deduction. In: Herbsleb, J.D., Dwyer, M.B. (eds.) Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014. pp. 155–166. ACM (2014). https://doi.org/10.1145/2593882.2593899, https://doi.org/10.1145/2593882.2593899

A Model Counting

Model counting is the counting of satisfying assignments (models) of a given (propositional) formula. We use model counting to quantify the number of inputs for which a given variable corresponds to our specification. This section gives a brief overview over the notions of model counting and recent advances in the field before it explains how we harness propositional model counting techniques in our pipeline.

Preliminaries Assuming a CNF formula ϕ over the signature Σ, an assignment is the interpretation of the propositional atoms $I \colon \Sigma \to \{t, f\}$. There are $2^{|\Sigma|}$ possible assignments for the variables in ϕ. We can then define $\text{models}(\phi) \coloneqq \{I \mid I \models \phi\}$ as the set of all assignments satisfying ϕ. A model counter thus computes the cardinality $|\text{models}(\phi)|$. In our case, we are only interested in the assignments of the program's input variables. For this we need the projection on propositional formulae. Let $\Delta \subseteq \Sigma$ be a signature, then $\phi{\downarrow_\Delta}$ denotes the strongest formula over Δ which is entailed by ϕ [12, Logical Foundations]. Note, $\phi{\downarrow_\Delta}$ states the same constraints on the atoms in Δ as ϕ. We denote with $\text{models}_\Delta(\phi) \coloneqq \{I{\downarrow_\Delta} \mid I \models \phi{\downarrow_\Delta}\}$ the set of Δ-models of the Δ-projection of ϕ. There exist exact and approximative model-counters.

Exact Model Counting. In our work we use the probabilistic exact model counter GANAK [17] which is freely available and represents an improved version of sharpSAT [22]. For a given parameter δ the model counter returns the exact model count with probability $1 - \delta$. In practice the tool returned exact results for all benchmarks considered by [17] when setting $\delta = 0.05$. The model counter also recently won a first place in the Model Counting Competition 2020 [9] as part of a portfolio.

Approximate Model Counting. Alternatively, we can use the approximate model counter ApproxMC [3,12,18,15] which uses a *probably approximate correct* (or PAC) algorithm. This means the algorithm provides a theoretical guarantee on the relation between the correct model count ψ and the result of ApproxMC $\psi_{\mathcal{A}}$ which can be configured by some (ϵ, δ):

$$\Pr\left[\frac{\psi}{1+\epsilon} \leq \psi_{\mathcal{A}} \leq (1+\epsilon)\psi\right] \geq 1 - \delta \tag{PAC}$$

It is worth noting that two runs of ApproxMC can be considered independent random variables as the correctness of the result depends on the random variables within the algorithm and not its input value.

Bounds for ratios. For the ratios $r^e_\forall$, and $r^f_\forall$ we obtain error bounds through the following corollary:

Corollary 1. *Let ϕ_1, ϕ_2 be two model counts measured according to (PAC) and let further ϕ_1^*, ϕ_2^* be the correct model counts, then:*

$$\Pr\left[1 - (1+\epsilon)^2 \frac{\phi_2^*}{\phi_1^*} \leq \frac{\phi_1 - \phi_2}{\phi_1} \leq 1 - \frac{\phi_2^*}{(1+\epsilon)^2 \phi_1^*}\right] \geq (1-\delta)^2$$

For both ratios ϕ_1 in Corollary 1 corresponds to $\psi_?^h$ (consequently ϕ_1^* represents $H_?$). ϕ_2 then either corresponds to $\psi_!^h$ or $\psi_!^m$ respectively yielding the measurement for $r_\forall^e$ or $r_\forall^f$ (hence ϕ_2^* represents $H_!$ or $M_!$). We clearly see that the ratios' error bounds become less precise in comparison to the single measure case.

B Correctness of the pipeline

In this section we will argue why the pipeline we described in Sec. 3 is correct. By correctness, we mean that we get correct counts $\psi_?^m, \psi_?^h, \psi_!^m$ and $\psi_!^h$ for $M_?, H_?, M_!$ and $H_!$ when running the pipeline with an exact model counter. The correctness depends on three elementary properties: First, the program transformation preserves the behavior of the original program. Of course, a new program flow is established, but in cases without a violation of an assumption or assertion the transformed program behaves equally. Secondly, every hit or miss of an assumption or assertion is captured faithfully by a violated assert-statement (Lines 20 to 23). Thirdly, every (projected) model of the generated CNF formulae is indeed a representative for a violating or valid program path.

Program transformation preserves behavior. We need to consider that unifying return statements, and early execution abortion do not alter the program flow unfaithfully, especially, no program behavior is introduced which violates an assumption or assertion.

Considering the unifying of return statements: When in the old program a return statement is reached, the sub-routine is immediately returned to the callee. In the transformed program, we store the return value if necessary, and jump to the last remaining return statement under the `end`-label. This takes an extra step, but this step does not modify the value of any program variable.

The early abortion is triggered only when an assumption or assertion violation occurs, as the variables `am` and `as` are not in the original program. Therefore, for any input without a specification violation, the transformed program behaves exactly the same, terminating with `am` and `as` equal to `false`.

Capturing hits and misses. We need to consider four cases (assumption/assert hit/miss), but due to symmetry reason, we only argue in the two critical cases: the violation of an assumption or assertion. Please reconsider, that we encode the required property with a negation into the program, because a negation on the property is added by CBMC. As a summary, we count the inputs which violate an assert-statement.

Let us consider the case of assumption misses ($M_?$). As previously explained, the resulting program should contain exactly one assert-statement which is violated iff the given input violates one of the specified assumptions. This assert-statement (see Line 23 in Fig. 5) checks the program variable `am`. The program execution, just as in the original program, begins with the subroutine under consideration and the checks of the assumptions. If at least one assumption is violated for a specific input, the variable `am` becomes `true`, and the program is aborted early. For the entry routine, we directly jump to the assert-statement which will be violated. Therefore, any violation of user-defined assertion is ignored in the further program flow.

Next, we consider the case of assumption hit ($H_?$) with an occurring assertion miss ($M_!$). The original (and transformed) program starts again in the entry function, but as the assumptions are met, the remaining body is executed (`am` stays `false` for the complete remaining execution). Afterwards, if an assertion in is violated somewhere during the program execution, the variable `as` becomes and stays `true`, and also the program flow of the current routine and its callees is aborted recursively until one of the assert-statements (Lines 20 to 23) is reached. In detail, only the assert-statements in Line 21 (assertion miss) and in Line 22 (assumption hit) are violated. If no assertion had been violated, the assert-statement in Line 20 would be violated.

Model-counting. Our projected model counter computes the number of distinguishable assignments projected to a variable set Δ which satisfy the given formula. If we want to obtain correct results for our metrics, we thus need to choose our formulae and projection set in such a way that the model counter returns correct counts. Our bounded model checker CBMC returns formulae which are satisfiable iff a specified assert-statement can be missed for some program path (within the defined bound). The formula returned by CBMC contains propositional variables for all bits of our program's input. We thus choose exactly those input representatives as set Δ. If we are able to construct a program containing exactly one assert-statement which is missed iff the given input has a program path that implies an assertion hit, we can then compute $H_!$ by constructing this program and subsequently passing the transformed program to CBMC and later on to the model counter projecting on the input variables. Correspondingly, we can compute $M_!, H_?$ and $M_?$ if we can construct a program containing exactly one assert-statement which is missed iff the conditions for the corresponding variable are met.