Symbolic Execution in CPAchecker

Thomas Lemberger
lemberge@fim.uni-passau.de

1. MOTIVATION

With the rise of ubiquitious computing, reactive software systems that constantly receive input from the outside become more and more common. Because of their non-functional behaviour, the unreliability of testing becomes even more severe for such programs. In constrast to this, automatic software verification offers a more reliable alternative by analyzing all possible behaviours of a program. Configurable software verification[2] and its implementation CPAchecker[3] represent a recent approach to software verification that has been successful in multiple iterations of the Competition on Software Verification (SV-COMP).

A prominent configurable program analysis (CPA) used in CPAchecker is the value analysis CPA [4]. But while it offers high efficiency, it cannot handle non-deterministic values. Consider the simple example program in Listing 1 and its analysis by the value analysis CPA in Figure 1. It can easily be seen that the non-deterministic value is handled as an unknown value and as such is discarded. Because of this over-approximation, necessary information about the relation between a and b gets lost and the safety property in line 14 is seen as violated.

This characteristic results in a high amount of false positives. In practice, this problem is countered by the use of counterexample checks and strengthening through other CPAs. Instead of that, we will introduce symbolic values to the existing value analysis CPA to allow it tracking of non-deterministic values. In addition, we specify a new constraints CPA to track constraints on these symbolic values.

First, we will give the formal specification of a composite CPA (which we will call symbolic execution CPA) using these two components. After this, an implementation of this CPA based on CPAchecker and its performance based on test sets of SV-COMP 2015 will be presented.

2. SPECIFICATION

```
1  extern __nondet_int();
2
3  int main() {
4    int a = __nondet_int();
5    int b;
6
7    if (a >= 0) {
8      b = a;
9
10   } else {
11     b = a + 1;
12   }
13
14   if (b < a) {
15 ERROR:
16     return -1;
17   }
18 }
```

Listing 1: A simple non-deterministic program

The symbolic execution CPA is a composite CPA of a *location CPA*, a *symbolic value CPA* which tracks and computes the deterministic and non-deterministic values of variables and a *constraints CPA*, which tracks encountered assumptions in form of constraints for each location's variables. The formal definitions of a composite CPA and location CPA are used as defined in [2].

For the sake of formal simplicity, we assume that all values are integers.

2.1 Symbolic value CPA

The described CPA is an extension of the priorly mentioned value analysis CPA. Instead of using a bottom element $\bot_{\mathbb{Z}_{\mathbb{CO}}}$, we represent an unreachable state through the absence of a valid transfer. The symbolic value CPA is defined as

$$\mathbb{S} = (D_{\mathbb{S}}, \rightsquigarrow_{\mathbb{S}}, merge^{sep}\ stop^{sep})$$

with abstract domain $D_{\mathbb{S}}$, transfer relation $\rightsquigarrow_{\mathbb{S}}$, merge operator $merge^{sep}$ and stop operator $stop^{sep}$.

The abstract domain $D_{\mathbb{S}}$ is defined as $D_{\mathbb{S}} = (C, \mathcal{E}, [\![\cdot]\!])$ with C being the set of concrete program states, $\mathcal{E}$ the semi-lattice of possible abstract states and $[\![\cdot]\!]$ the concretization function.

$$\mathcal{E} = (V_{\mathbb{S}}, \sqsubseteq, \sqcup, v_{\top})$$

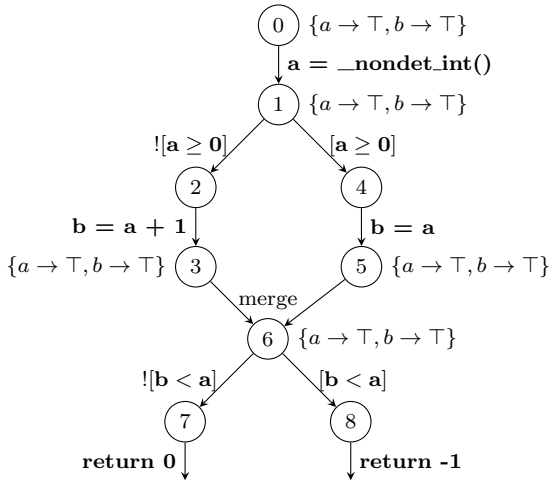

Figure 1: Analysis of the program in Listing 1 by the value analysis CPA. The current state of the CPA is noted after every assignment edge.

The elements of the semi-lattice are total functions of $V_{\mathbb{S}} = X \to (\mathcal{Z}_{\mathbb{CO}} \cup \mathcal{Z}_{\mathbb{S}})$ mapping program variables X to a concrete value $\mathcal{Z}_{\mathbb{CO}} = \mathbb{Z} \cup \top_{\mathcal{Z}}$ or to a symbolic value $\mathcal{Z}_{\mathbb{S}} = S_I \cup S_E$. S_I describes all symbolic identifiers and S_E is the set of all valid symbolic expressions. Each expression is a symbolic expression, if at least one symbolic identifier occurs in it. $\top_{\mathcal{Z}}$ represents an unknown value.

The lesser-than-or-equal operator $\sqsubseteq$ of $\mathcal{E}$ is defined by

$$v \sqsubseteq v' \Leftrightarrow \forall x \in X : v(x) = v'(x) \vee v'(x) = \top_{\mathcal{Z}}.$$

The join $\sqcup$ holds the least upper bound of two lattice elements $v, v' \in V_{\mathbb{S}}$ with

$$(v \sqcup v')(x) = \begin{cases} v(x) & \text{if } v(x) = v'(x) \\ \top_{\mathcal{Z}} & \text{otherwise} \end{cases}$$

.

The top element of $\mathcal{E}$ is defined as $v_\top(x) = \top_{\mathcal{Z}}$ for all $x \in X$.

A concrete state $c \in C$ is *compliant* to an abstract variable assignment v, if an arbitrary, but valid assignment of concrete values to symbolic identfiers $d : S_I \to \mathbb{Z}$ exists, so that for all $x \in X$ one of the following conditions is fulfilled: (1) the abstract assignment is a concrete value that equals the value in the concrete state, that means $c(x) = v(x)$, (2) the abstract assignment $v(x)$ is a symbolic value that can be evaluated to $c(x)$ by replacing all occurring symbolic identifiers $i \in S_I$ with $d(i)$, or (3) x has not occurred yet and as such has no concrete or symbolic value assigned, represented by $v(x) = \top_{\mathcal{Z}}$. The concretization function $[\![\cdot]\!]$ then assigns all compliant concrete states to an abstract state v.

The transfer relation $\rightsquigarrow_{\mathbb{S}}$ contains the transfer $v \overset{g}{\rightsquigarrow} v''$, if one of the following conditions is true:

1. $g = (l, assume(p), l')$, $\phi(p, v)$ is satisfiable and for all $x \in X$:

$$v''(x) = \begin{cases} c & \text{if } c \text{ only satisfying assignment for } \phi(p, v) \\ y & \text{if } v(x) = \top_{\mathcal{Z}} \text{ and } x \text{ appears in } p.\ y \in S_I \text{ and y is a new value that has not been used in any other state before.} \\ v(x) & \text{otherwise} \end{cases}$$

with

$$\phi(p, v) := p \wedge (\bigwedge_{\substack{x \in X, \\ v(x) \in \mathbb{Z}}} x = v(x))$$

Note: ϕ performs an over-approximation, as only variables $x \in X$ with an explicit value are considered.

2. $g = (l, w := e, l')$ and for all $x \in X$:

$$v''(x) = \begin{cases} eval(e, v') & \text{if } x = w \\ v'(x) & \text{otherwise} \end{cases}$$

with

$$v'(x) = \begin{cases} y & \text{if } v(x) = \top_{\mathcal{Z}} \text{ and } x \text{ appears in } e.\ y \in S_I \text{ and y is a new value that has not been used in any other state before.} \\ v(x) & \text{otherwise} \end{cases}$$

and $eval(e, v')$ defined as the evaluation of an expression e with the given assignment. If any symbolic value occurs in $v(e)$, it is only partially evaluated. In this case, $eval(e, v') \in \mathcal{Z}_{\mathbb{S}}$.

3. $v'' = v_\top$.

The merge operator $merge^{sep}$ is defined as $merge^{sep}(e, e') = e'$. The stop operator $stop^{sep}$ is defined as $stop^{sep}(e, R) = \exists e' \in R : e \sqsubseteq e'$

2.2 Constraints CPA

The constraints CPA is a CPA

$$\mathbb{C} = (D_{\mathbb{C}}, \rightsquigarrow_{\mathbb{C}}, merge^{sep}, stop^{sep})$$

that tracks constraints (i.e. boolean formulas) on variables that are created by assume edges. The abstract domain $D_{\mathbb{C}}$ is defined by

$$D_{\mathbb{C}} = (C, \mathcal{C}, [\![\cdot]\!])$$

with concrete states C, the semi-lattice $\mathcal{C}$ and concretization function $[\![\cdot]\!]$.

The abstract states described by $\mathcal{C} = (2^\gamma, \sqsubseteq, \sqcup, \top)$ consist of constraints of γ. γ contains all possible boolean expressions over $\mathcal{Z}_{\mathbb{CO}} \cup \mathcal{Z}_{\mathbb{S}}$, including symbolic expressions. An abstract state is interpreted as the conjunction of all its constraints.

$\sqsubseteq$ is defined as $a \sqsubseteq a' \Leftrightarrow a' \subseteq a$, the join $\sqcup$ as the intersection: $a \sqcup a' = a \cap a'$

The concretization function $[\![\cdot]\!]$ maps an abstract state to all concrete states that satisfy this abstract state's constraints.

$$[\![a]\!] = \{c \in C |\ c \vDash \bigwedge_{\varphi \in a} \varphi\}$$

Note: We assume that the empty conjunction represents true, that means $\bigwedge_{\varphi \in \{\}} \varphi = true$.

Constraint CPA's transfer relation $\rightsquigarrow_{\mathbb{C}}$ contains the transfer $a \overset{g}{\rightsquigarrow}_{\mathbb{C}} a'$ if one of the following conditions is fulfilled: (1) $g = (l, assume(p), l')$, $a' = a \cup p$ and a does not contain any variable $x \in X$ (By specifying that a must not contain any variable we enforce that variables are always replaced by concrete (explicit or symbolic) values through strengthening), or (2) $a' = a$ otherwise.

As p is only added as a constraint if no variable of X occurs in a, all resulting states of this transfer relation will be satisfiable, unless p is a contradiction ($a \neq a$, for instance). Because of this, we completely disregard checking for satisfiability of constraints here. Instead, we rely on satisfiability checks during strengthening by other CPAs. In the following we will use the priorly defined symbolic value CPA to replace newly added constraints' variables with values and check for the whole state's satisfiability.

2.3 Composition of CPAs

$\mathbb{C}_{\mathbb{LSC}} = (\mathbb{L}, \mathbb{S}, \mathbb{C}, \rightsquigarrow_x, merge^{sep}, stop^{sep})$ is the composite CPA of a location CPA and the priorly defined symbolic value CPA and constraints CPA. The transfer relation

$$\rightsquigarrow_x : D_{\mathbb{L}} \times D_{\mathbb{S}} \times D_{\mathbb{C}} \to D_{\mathbb{L}} \times D_{\mathbb{S}} \times D_{\mathbb{C}}$$

contains the transfer $(l, v, a) \overset{g}{\rightsquigarrow}_x (l', v', a'')$ if $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$, $v \overset{g}{\rightsquigarrow}_{\mathbb{S}} v'$ and $a \overset{g}{\rightsquigarrow}_{\mathbb{C}} a'$ and if the strengthen operation $\downarrow_{\mathbb{C},\mathbb{S}}$ is defined for a' and v' with $\downarrow_{\mathbb{C},\mathbb{S}} (a', v') = a''$.

The strengthen operator $\downarrow_{\mathbb{C},\mathbb{S}} : \mathbb{C} \times \mathbb{S} \to \mathbb{C}$ uses the value assignment $v(x)$ of the given second operand to replace variables of the latest, unstrengthened constraint of the given constraints state. This way, meaningful constraints are created that show relations between symbolic values and can be checked for satisfiability.

$\downarrow_{\mathbb{C},\mathbb{S}} (a, v) = a'$ is defined if a' is the result of replacing all program variables $x \in X$ occurring in a with $v(x)$ and if $\bigwedge_{\varphi \in a'} \varphi$ is satisfiable.

An analysis of the previous test program using symbolic execution can be seen in figure 2. By storing information about the relationship of variables a and b, the CPA can derive at location 6/6' that b can't be smaller than a.

3. EVALUATION

The specified CPA was implemented in CPAchecker[3] using the existing CompositeCPA and LocationCPA. The existing ValueAnalysisCPA was extended to fulfil the specification of our symbolic value CPA, while the constraints CPA was implemented as a new, own CPA. To check the satisfiability of constraints, an external SAT checker is used. For these benchmarks, MathSAT5 is used with the bitvector theory, encoding float values as floats. The analysis is part of CPAchecker's trunk since revision 16052 and can be activated by using the configuration `valueAnalysis-symbolic`. As development is ongoing, the current revision at the time of this writing, that is 16071, was chosen for benchmarks.

	Value	SymEx	Overall
correct	1994	475	3038
true positives	631	322	802
true negatives	1363	153	2236
unique true positives	318	8	-
unique true negatives	1243	33	-
false positives	271	56	-
unique false positives	220	5	-
false negatives	0	0	-
timeouts	662	2451	-
errors	111	56	-

Figure 3: Result of runs of value analysis (Value) and symbolic execution (SymEx) on SV-COMP 2015 test sets. "true positive" represents a found error in a program with an error, "true negative" represents no found error in a program without errors.

Benchmarks were performed on a subset of the SV-COMP 2015 test set, excluding sets CPAchecker's ValueAnalysis-CPA has no competency in. These excluded sets are namely "Concurrency", "Memory Safety", "Recursive" and "Termination". An overview of all test sets can be found at [1]. Tests were executed on an Intel Xeon E7-4870 at 2.40GHz with a memory limit of 15GB and a time limit of 900 seconds.

Figure 3 shows the performance of the default ValueAnalysisCPA without counterexample-guided abstraction refinement (CEGAR) or counterexample checks next to the performance of our symbolic execution configuration. The category "timeouts" does not only include timeouts, but also two StackOverflowExceptions that resulted from long symbolic expressions. Such expressions can be generated by long loops, but are uncommon. Generally speaking, basic value analysis easily outperforms symbolic execution in numbers. Lots of timeouts occur due to (a) path explosion, an exponential increase in possible paths based on the number of branches (that is if-statements and loops) as a result of the high precision of symbolic execution, and (b) the bad performance of SAT checks with a large number of variables and arithmetic operations like multiplication. Path explosion can easily been illustrated when looking at the algorithm statistics of a simple program using multiple if- and goto-statements: While basic value analysis only needs about 3500 iterations to find a possible error, with the biggest waitlist consisting of 14 entries, analysis with symbolic execution needs over 25000 iterations with the biggest waitlist consisting of almost 530 states. In addition to this enormous growth in iterations, SAT checks are responsible for up to 95% of CPU time in analyses of programs that are *not* specifically constructed for challenging SAT checkers.

Despite this reduced speed, symbolic execution's ability to track non-deterministic values allows it to get a correct result in 41 cases in which value analysis does not. In eight of these value analysis receives timeouts, and in 33 value analysis provides a false positive. Symbolic execution does never produce a false result when value analysis produces a correct one.

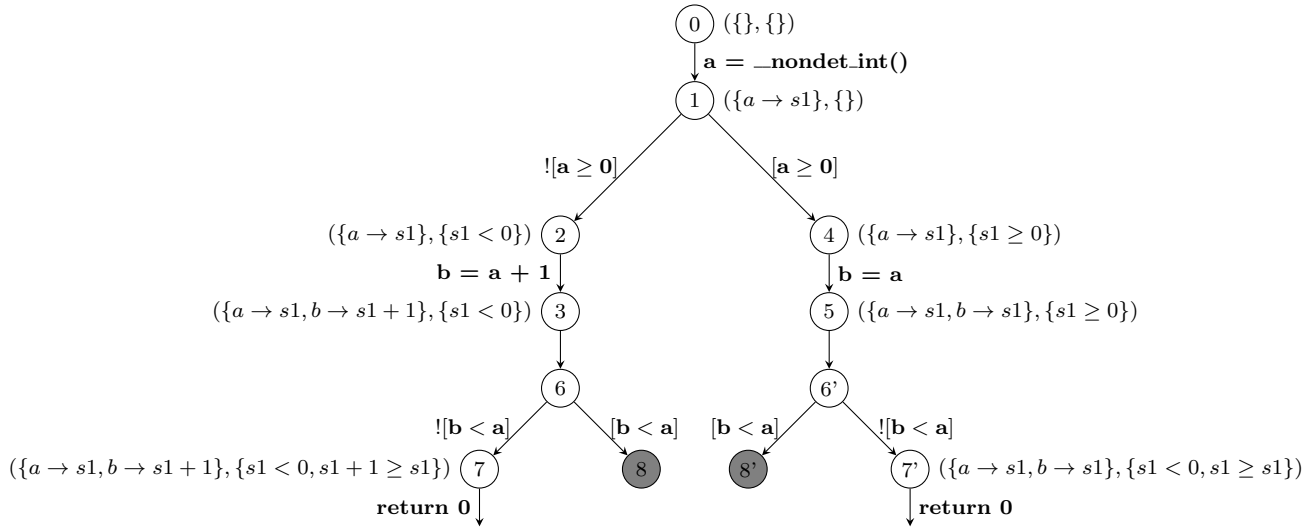

Figure 2: Analysis of the program in listing 1 by the symbolic execution CPA. The locations reported as unreachable are tinted grey. The state is provided as a tuple at each relevant node, the first element being the state of $\mathbb{S}$, the second of $\mathbb{C}$. The state of the location CPA can be derived from the node's numbers.

	Bitvector theory	Integer theory
correct	475	461
false positives	56	97
false negatives	0	6
timeouts	2450	2417
errors	56	56

Figure 4: Results of symbolic execution CPA using bitvector theory with floats and integer theory on SV-COMP 2015 test sets.

When using a timelimit of 1800 seconds, symbolic execution is able to compute 14 more results than with a timelimit of 900 seconds, of which eleven are correct.

Of these long-taking runs, five are the result of path explosion, while the other nine only consist of few iterations, but contain expressions resulting in long-taking SAT checks, namely bitvector computations, non-deterministic floats and multiplications.

Some of these performance issues can be improved by using another theory in SAT checks. By using an integer theory instead of bitvector, runtime could be decreased by up to 50%, but resulted in a worse overall performance. The use of integer results in an unsound analysis, too. Figure 4 shows the difference between the use of a bitvector theory with floats and the use of an integer theory.

Because of the obviously worse performance of integer theories, a bitvector theory should be used. Instead, it should be tried to decrease runtime by tackling the problem of path explosion. A valid approach to this could be CEGAR, which is already implemented for the ValueAnalysisCPA and is described in [4]. By resolving the problem of path explosion, the number of iterations and as such also the number of SAT checks could be decreased. This should be the main focus of future work.

4. REFERENCES

[1] Beyer, D. Competition on Software Verification (SV-COMP) 2015: Benchmark Verification Tasks, 2015.

[2] Beyer, D., Henzinger, T. a., and Théoduloz, G. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. *Computer Aided Verification 4590* (2007), 504–518.

[3] Beyer, D., and Keremoglu, M. E. CPAchecker: A tool for configurable software verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6806 LNCS* (2011), 184–190.

[4] Beyer, D., and Löwe, S. Explicit-Value Analysis Based on CEGAR and Interpolation. Tech. Rep. December, Department of Computer Science and Mathematics, University of Passau, Passau, 2012.