Ultimate Automizer and the Abstraction of Bitwise Operations
(Competition Contribution)

Frank Schüssele, Manuel Bentele, Daniel Dietsch, Matthias Heizmann*, Xinyu Jiang, Dominik Klumpp, and Andreas Podelski

University of Freiburg, Freiburg im Breisgau, Germany
schuessf@informatik.uni-freiburg.de

Abstract. The verification of Ultimate Automizer works on an SMT-LIB-based model of a C program. If we choose an SMT-LIB theory of (mathematical) integers, the translation is not precise, because we overapproximate bitwise operations. In this paper we present a translation for bitwise operations that improves the precision of this overapproximation.

1 Verification Approach

Ultimate Automizer (in the following abbreviated as UAutomizer) is a software verifier that implements the trace abstraction approach [6,9]. In trace abstraction, a verification problem is considered as a formal language and decomposed via automata-theoretic methods into smaller verification problems. While verifying a C program, UAutomizer applies trace abstraction to a model of the program that consists of a control-flow graph (CFG) and SMT-LIB formulas that express how the program's data is modified while moving along an edge of the CFG. We obtain this model by first translating the C program into a Boogie [10] program and afterwards translating the Boogie program into the CFG and SMT-LIB formulas. We have two variants of these translations, we call them the *integer-based translation* and the *bitvector-based translation*. The integer-based translation results in a Boogie program over mathematical integers that is later translated to SMT-LIB formulas from the integer theory. The bitvector-based translation results in a Boogie program over bitvectors that is later translated to SMT-LIB formulas from the bitvector theory. The integer-based translation uses modulo operations to make sure that the result of arithmetic operation is in the correct range. It also overapproximates the result of bitwise operations and is hence not very precise. If the trace abstraction-based verification algorithm returns a counterexample that contains an overapproximated operation, UAutomizer does not return the counterexample but *unknown* instead. The bitvector-based translation returns a result that is precise but whose verification is costly. In order to mitigate the shortcomings of both translations, UAutomizer first runs the verification on the integer-based model. If the result is *unknown*, the tool is run again on the bitvector-based model.

* Jury Member: Matthias Heizmann

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 418–423, 2024.
https://doi.org/10.1007/978-3-031-57256-2_31

2 Abstraction of Bitwise Operations

In the past our integer translation overapproximated the bitwise operators, i.e. `&`, `|`, `^`, `~`, `<<`, `>>` returned some non-deterministic value. In this paper we show how to translate bitwise operators more precisely. Our translation is a generalization of the work of Liu et al. [11]. First we describe the translation of the operators `&`, `|`, `^`. The remaining operators will be explained at the end of this section. For the operators `&`, `|`, `^` we distinguish three different cases:

– If both operands are literals, we replace the operation by its result.
– If one operand is a literal with a specific bit-pattern, we rewrite the expression directly.
– Otherwise we overapproximate with additional constraints for the return value.

Rewrite rules. If one of the operands is a literal, we try to replace the bitwise operation by an arithmetic operation based on the bit-pattern of the literal. These rewrite rules are shown in Table 1 (omitting symmetric cases). The first two cases are simple. In the first row every bit is zero (i.e. the operand is 0). Zero is the absorbing element for `&` and the neutral element for `|` and `^`. In the second row every bit is one (i.e. the operand is -1 for signed integers or the maximum value for unsigned integers). This is the neutral element for `&` and the absorbing element for `|`. The last two cases are motivated by typical bitmasks and are a generalization of the first two cases. In a C program, bitmasks are used to set bits to zero or to one. For example the expression `x & 255` can be used to replace every bit of `x` by zero except for the last 8 bits. The third row is motivated by Liu et al. [11]. They rewrote `x & 1` (i.e. only the last bit is one) to `x % 2`, whereas we generalize this case for any pattern that only ends with ones. With the rule on the third row the expression `x & 255` is rewritten to `x % 256`. In the last row only the starting bits are one. This case works analogously to the third row, it is rewritten using a combination of modulo and other arithmetic operators. We implemented these rules in our translation from C to Boogie. Boogie has mathematical integer semantics, so the evaluation of the expressions in the table can never lead to an overflow. The rules for the operators `|` and `^` are based on the equalities $a \mid b = a + b - (a \mathbin{\&} b)$ and $a \mathbin{\hat{}} b = a + b - 2 \cdot (a \mathbin{\&} b)$.

Constrained Overapproximation. If none of the operands are literals with a bit-pattern from above, we translate the bitwise operations to calls to functions as implemented in Fig. 1 in Boogie as follows: `x & y` is translated to `and(x, y)`, `x | y` is translated to `or(x, y)` and `x ^ y` is translated to `xor(x, y)`. We omitted

Table 1: Rewrite rules based on the bit-pattern of `c`

`bits(c)`	`x & c`	`x | c`	`x ^ c`
`0...0`	`c`	`x`	`x`
`1...1`	`x`	`c`	`c - x`
`0...01...1`	`x % (c + 1)`	`x + c - x % (c+1)`	`x + c - 2*(x % (c + 1))`
`1...10...0`	`x - x % (c + 1)`	`c + x % (c + 1)`	`c - x - 2 * (x % (c + 1))`

```
procedure and(a: int, b: int) {
  if (a == 0 || b == 0) return 0;
  if (a == b) return a;

  var r: int;
  assume (a>=0 || b<0) ==> r<=a;
  assume (a<0 || b>=0) ==> r<=b;
  assume (a>=0 || b>=0) ==> r>=0;
  assume (a<0 || b<0) ==> r>a+b;
  return r;
}
```

```
procedure xor(a: int, b: int) {
  if (a == 0) return b;
  if (b == 0) return a;
  if (a == b) return 0;

  var r: int;
  assume (a>=0 <==> b>=0) ==> r>0;
  assume !(a>=0 <==> b>=0) ==> r<0;
  assume (a>=0 || b>=0) ==> r<=a+b;
  return r;
}
```

Fig. 1: Procedures to overapproximate the operators `&` and `^`

the definition for the function `or(a, b)` here, because a possible implementation could simply use the relation between `&` and `|` to return `a + b - and(a, b)`. The first lines of `and` and `xor` cover the cases that are handled precisely, i.e. where one of the operands is zero or both are equal. For all other cases return a non-deterministic value to overapproximate the behavior of the bitwise operators. We constrain this value via the assumptions that often provide lower and upper bounds. For example, if `a` and `b` are both non-negative, `and(a, b)` returns also a non-negative value that is also smaller or equal to both `a` and `b`. Similarly `xor(a, b)` returns a positive value that is smaller or equal to the sum `a + b` in that case.

Negation and Shifts. We rewrite the negation ∼`x` to the equivalent expression `-1 - x`. We rewrite shift operators if the second operand is a literal. The left shift `x << y` is rewritten to `x * c` and the right shift `x >> y` is rewritten to `x / c`, where `c` is the literal that is obtained by evaluating `pow(2, y)`. The rewritten expression `x * c` has an overflow if and only if the original expression `x << y` has an overflow.

3 Strengths and Weaknesses

UAutomizer won the overall category and the category NoOverflows in SV-COMP 2024 [2]. UAutomizer reported 10 incorrect results, which were due to incorrect modelling of C features.

We evaluated the abstraction of bitwise operations on selected benchmarks from SV-COMP 2024. The evaluation was performed on a AMD Ryzen Threadripper 3970X using 2 cores at 3.7 GHz with a time limit of 900 s and a memory limit of 8 GB. In Table 2 you can see the results of the evaluation on the category ReachSafety. We choose this category, because it contains a wide range of benchmarks, including several that make use of bitwise operators. There we compared three settings: the bitvector-based translation, the old integer-based translation where every bitwise operation is allowed to return any value and the integer-based translation with the optimizations described in Section 2. The results show that the new integer-based translation can verify 25 more benchmarks than the old integer-based translation (from various folders, e.g. hardness-nfm22

Table 2: Comparison on ReachSafety

	Bitvector			Integer (optimized)			Integer (old)		
	#	time (h)	mem (GB)	#	time (h)	mem (GB)	#	time (h)	mem (GB)
total (10 205)	1 958	65	1 862	2 076	37	2 600	2 051	36	2 550
safe (7 557)	1 183	41	1 030	1 350	22	1 510	1 324	21	1 440
unsafe (2 648)	775	24	832	726	15	1 090	727	15	1 110

Table 3: Comparison on Termination-BitVectors

	Integer (optimized)			Integer (old)		
	#	time (s)	mem (GB)	#	time (s)	mem (GB)
total (37)	31	410	12.1	12	122	4.2
safe (23)	23	325	9.2	7	73	2.5
unsafe (14)	8	85	2.9	5	49	1.7

and hardware-verification) and 118 more than the bitvector-based translation. The bitvector-based translation is precise in contrast to the integer-based translation. Overall this precision does not pay off, as the result of the bitvector-based translation is often too costly to verify. However, the precision can also be helpful, as the bitvector-based translation can find 48 (resp. 49) more bugs than the integer-based translations.

We also evaluated our approach on the subcategory Termination-BitVectors, where most of the benchmarks contain bitwise operations. For termination we do not support bitvectors, therefore we compared only our approach with the old integer-based translation. The results in Table 3 show that the our optimized approach is sufficient to prove the (non-)termination of 31 of the total 37 tasks, whereas the trivial overapproximation is only sufficient for 12.

4 Architecture, Setup, Configuration, and Project

UAutomizer is part of Ultimate [15,16], a program analysis framework written in Java and licensed under LGPLv3. UAutomizer is an automaton-based model checker using a CEGAR-loop approach [8]. The submitted version 0.2.4-0e0057cc requires Java 11 and Python 3.6. Its Linux version, binaries of the required SMT solvers Z3 [12,13], CVC4 [1,14], MathSAT [4,7], and a Python wrapper script were submitted as a `.zip` archive. UAutomizer is invoked with

```
./Ultimate.py --spec <p> --file <f> --architecture <a> --full-output
```

where `<p>` is an SV-COMP property file, `<f>` is an input C file, `<a>` is the architecture (`32bit` or `64bit`), and `--full-output` enables verbose output to `stdout`. A witness is written to the files `witness.graphml` and `witness.yml`. The benchmarking tool BenchExec [3] supports UAutomizer through the tool-info module `ultimateautomizer.py`. UAutomizer participates in all categories, as declared in its benchmark definition file `uautomizer.xml`.

Data Availability. The competition contribution for UAutomizer is available as an archive on Zenodo [5].

References

1. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
2. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
3. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
4. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
5. Dietsch, D., Bentele, M., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Automizer SV-COMP 2024 Competition Contribution (Nov 2023). https://doi.org/10.5281/zenodo.10203545
6. Dietsch, D., Heizmann, M., Klumpp, D., Naouar, M., Podelski, A., Schätzle, C.: Verification of concurrent programs using Petri net unfoldings. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12597, pp. 174–195. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_9
7. Fondazione Bruno Kessler, D.: MathSAT, https://mathsat.fbk.eu, (retrieved 2024-02-12)
8. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018. Lecture Notes in Computer Science, vol. 10806, pp. 447–451. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
9. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: SAS. Lecture Notes in Computer Science, vol. 5673, pp. 69–85. Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
10. Leino, K.R.M.: This is Boogie 2 (June 2008), https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
11. Liu, Y.C., Pang, C., Dietsch, D., Koskinen, E., Le, T., Portokalidis, G., Xu, J.: Proving LTL properties of bitvector programs and decompiled binaries. In: Oh, H. (ed.) Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13008, pp. 285–304. Springer (2021). https://doi.org/10.1007/978-3-030-89051-3_16

12. Microsoft Corporation: Z3, https://github.com/Z3Prover/z3, (retrieved 2024-02-12)
13. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. Stanford University, U.: CVC4, https://cvc4.github.io, (retrieved 2024-02-12)
15. University of Freiburg: Ultimate source code repository, https://github.com/ultimate-pa/ultimate, (retrieved 2024-02-12)
16. University of Freiburg: Ultimate website, https://ultimate-pa.org, (retrieved 2024-02-12)