C32SAT: Checking C Expressions
(Tool Paper)

Robert Brummayer and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria
{robert.brummayer,armin.biere}@jku.at

Abstract. C32SAT is a tool for checking C expressions. It can check whether a given C expression can be satisfied, is tautological, or always defined according to the ISO C99 standard. C32SAT can be used to detect nonportable expressions where program behavior depends on the compiler. Our contribution consists of C32SAT's functional representation and the way it handles undefined values. Under-approximation is used as optimization.

1 Introduction

Formal verification of C programs is an active area of research [6,7,8,11]. C32SAT[1] addresses a verification problem not explicitly considered by other verification tools. It detects situations where, according to the C99 standard [9], the behavior upon an operation on certain values is undefined, e.g. the behavior upon dividing an integer by zero. The C99 standard [9] describes undefined behavior as "behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements" [9]. The execution of such an undefined operation ranges from ignoring the situation to terminating the execution in the worst case.

In contrast to other programming languages, e.g. Java, there are many cases in the C programming language where undefined behavior can occur. This situation makes it hard to write secure and portable programs where the behavior is fully defined and does not depend on compiler semantics.

If the behavior upon an operation is undefined, then C32SAT raises a flag that marks the result to be *undefined*. This flag propagates and can only be masked out by short circuit evaluation of the logical conjunction `&&`, logical disjunction `||` and the conditional operator `?:`. Note that except for these three operations the order of evaluation of subexpressions is undefined as in the C99 standard.

C32SAT takes as input one C expression. It can check whether it can be satisfied, is tautological or always defined according to the C99 standard. C32SAT supports all main C operators, including multiplication, division and modulo. Additionally, C32SAT supports logical implication and equivalence. Pointer related operators are scheduled as future work.

[1] http://fmv.jku.at/c32sat/

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 294–297, 2007.

2 System Architecture

The core of C32SAT version 1.4 consists of approximately 7500 lines of C code. Figure 1 shows the core components of C32SAT. The frontend mainly consists of the components `Parser` and `Parse Tree`. The remaining components are part of the backend. The architecture is similar to that of a compiler except that the backend generates a Conjunctive Normal Form (CNF) instead of machine code.

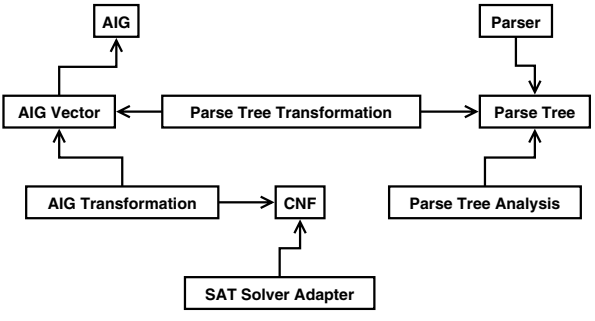

Fig. 1. Core architecture of C32SAT

3 Internal Functionality

C32SAT treats the type of every variable as *signed* integer. The bit width $w \in \{4, 8, 16, 32, 64\}$ can be globally configured. A variable is internally represented by a vector of And-Inverter Graphs (AIGs) [10] where every AIG represents exactly one bit.

Each C operator is mapped to a circuit that takes the AIG vector operands as input. For example the result of the word level XOR operator `^` is an AIG vector where the AIG vectors of the operands are bitwise combined by boolean XOR. This functional representation is in contrast to COGENT [8], which uses a relational representation. Our approach allows the application of sophisticated circuit simplification techniques like local two-level AIG rewriting [4] and the application of structural SAT solvers.

Every integer is actually represented by $w + 1$ AIGs. The additional AIG represents the *undefined* value. In general an expression is undefined when a subexpression is undefined. The only exception is short-circuit evaluation of `&&`, `||` and `?:`.

Regarding C32SAT's set of operators the result of every operation is either fully defined or fully undefined. It is never the case that only a part of the bits is undefined while the remaining bits are defined. Therefore, C32SAT handles undefined values on the AIG vector level and not on the AIG level.

The general flow of C32SAT is the following. C32SAT parses the input expression and builds a parse tree, which is analyzed and transformed into an AIG. Afterwards, the AIG is transformed into Conjunctive Normal Form (CNF) using

Tseitin Transformation [12] and passed to a SAT solver. Alternatively, the AIG can be dumped in the AIGER [2] format.

The default SAT solver of C32SAT is PicoSAT [2]. Additionally, C32SAT supports the SAT solvers NanoSAT [1], BooleForce and CompSAT [3] [3]. The SAT solver computes if the CNF is satisfiable or not and returns a model in the satisfiable case. C32SAT uses this model to generate a word level model, which is printed out as part of the result. As an example consider the C expression `y != 0 => x / y`. We want to determine if there exists an assignment to x and y, for which the result of the expression is undefined. C32SAT generates a corresponding CNF, which is passed to the SAT solver. If the SAT instance is unsatisfiable, then the result of the expression is always defined. However, if the SAT instance is satisfiable, then C32SAT can use the satisfying assignment to generate a useful counter example.

Actually, C32SAT shows that unrestricted division can lead to an overflow. If we divide `INT_MIN` by -1, then we get a signed integer overflow, because in two's complement the negation of `INT_MIN` is undefined, as the behavior upon signed integer overflow is undefined in the C99 standard.

Note that if we added the constraint `y != -1` to the premise of the implication, then the result of the expression would always be defined.

4 Under-Approximation Optimization

Inspired by [5], we added an under-approximation optimization technique to the latest version of C32SAT. Instead of encoding an n-bit integer variable with n AIGs, we simply restrict the number of AIGs used for encoding. For example we encode a 32 bit integer variable in the following way. We represent the least significant bit by one AIG variable and all other bits by another. This AIG vector represents the values from -2 to 1 instead of -2147483648 to 2147483647.

If the under-approximated SAT instance is satisfiable, then also the original formula is satisfied by the same assignment. However, if the under-approximated SAT instance is unsatisfiable, then the approximation has to be refined. In this case C32SAT doubles the precision of the under-approximation. In the worst case no satisfying assignment can be found during under-approximation and C32SAT has to generate the full CNF.

Using this under-approximation technique leads to smaller AIGs. This results in a smaller CNF, which is typically easier to solve. Beside speeding up the search for satisfying assignments, the under-approximation technique produces assignments that are easier to interpret.

5 Conclusion

We presented C32SAT, a tool for checking C expressions. It can be used to detect nonportable expressions where program behavior depends on the compiler.

[2] `http://fmv.jku.at/aiger/`

[3] The SAT solvers are available at `http://fmv.jku.at/software/`

We presented C32SAT's functional representation, the way it handles undefined values and its under-approximation optimization technique. As future work we want to support pointers and over-approximation techniques.

References

1. Biere, A.: The evolution from LIMMAT to NANOSAT. Technical Report 444, Dept. Computer Science, ETH Zürich (2004)
2. Biere, A.: Occurrence lists for 2-watched literal schemes (submitted)
3. Biere, A., Sinz, C.: Decomposing sat problems into connected components. Journal on Satisfiability, Boolean Modeling and Computation (JSAT), 2 (2006)
4. Brummayer, R., Biere, A.: Local two-level And-Inverter Graph minimization without blowup. In: Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06) (2006)
5. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Proceedings of TACAS 2007, Springer, Heidelberg (to appear)
6. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
7. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
8. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 296–300. Springer, Heidelberg (2005)
9. ISO/IEC. Programming languages - C (ISO/IEC 9899:1999(E)) (1999)
10. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans. on CAD of Integrated Circuits and Systems 21(12), 1377–1394 (2002)
11. Sethi, N., Barret, C.: Cascade: C assertion checker and deductive engine. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 166–169. Springer, Heidelberg (2006)
12. Tseitin, G.S.: On the complexity of derivation in the propositional calculus. Studies in constructive mathematics and mathematical logic, pp. 115–125 (1968)