Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution) ⋆

Fei He[1,2,3] (✉), Zhihang Sun[1,2,3], and Hongyu Fan[1,2,3]

[1] School of Software, Tsinghua University, Beijing, China
[2] Key Laboratory for Information System Security, MoE, Beijing, China
[3] Beijing National Research Center for Information Science and Technology, Beijing, China

Abstract. `Deagle` is an SMT-based multi-threaded program verification tool. It is built on top of `CBMC` (front-end) and `MiniSAT` (back-end). The basic idea of `Deagle` is to integrate into the SMT solver an ordering consistency theory that handles ordering relations over the shared variable accesses in the program. The front-end encodes the input program into an extended propositional formula that contains ordering constraints. The back-end is reinforced with a solver for the ordering consistency theory. This paper presents the basic idea, architecture, installation, and usage of `Deagle`.

Keywords: Program verification · Satisfiability modulo theories · Concurrency.

1 Verification Approach

Given a multi-threaded program, the thread communication behaviors can be modeled using the *happens-before* relations over memory access (read/write) events [1]. There are various kinds of happens-before relations: program order (PO), read-from order (RF), write serialization order (WS), and from-read order (FR). A *happens-before ordering formula* (abbreviated as *ordering formula*) is a logical formula that involves only memory access events and happens-before relations.

`Deagle` is an SMT-based multi-threaded program verifier, which consists of

- a front-end that encodes the intra-threaded behaviors (e.g., the control and data flow per thread) into propositional formulas, and the inter-threaded behaviors (i.e., the communication between threads) into *ordering formulas*;
- a back-end that extends `MiniSAT` with an *ordering consistency theory* solver [8] by following the DPLL(T) framework [7], and is able to solve propositional formulas and ordering formulas mixed together.

⋆ This work was supported in part by the National Key Research and Development Program of China (No. 2018YFB1308601) and the National Natural Science Foundation of China (No. 62072267 and No. 62021002).

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 424–428, 2022.
https://doi.org/10.1007/978-3-030-99527-0_25

Compared with [8]: The theory solver in [8] uses a *from-read axiom* to derive *FR* orders. Besides the *from-read axiom*, `Deagle` also implements a *write-serialization axiom* [11], with which *WS* orders can also be derived. In return, the front-end of `Deagle` need not encode both *FR* and *WS* orders explicitly.

2 Software Architecture

`Deagle` is developed on top of `CBMC` [9] and `MiniSAT` [6] using C++. Additionally, for ease of usage and debugging, `Deagle` reuses some modules developed in `Yogar-CBMC` [10,11]. `Deagle` is not a strategy selection-based verifier. `Deagle` runs the following procedures successively to verify a given C program:

Preprocessing (from `Yogar-CBMC`) For each global structure variable in the C program, the preprocessing procedure unfolds it by creating a fresh variable for each member. Note that arrays need no preprocessing; `CBMC` is able to handle each array as an entity.

Parsing and Goto-Program Generation (originally in `CBMC`) `CBMC` employs `Flex` and `Bison` to transform the preprocessed C program into an *abstract syntax tree* (*AST*). Then `CBMC` builds a *goto program*, where all branching statements and loop statements are represented with (conditional) goto statements.

Library Function Modeling (extended from `CBMC`) `CBMC` models each multithreading-related library function (e.g., *pthread_cond_wait*). For example, mutex m contains a Boolean variable *m_locked* indicating whether m is locked; *pthread_mutex_lock*(&m) assumes *m_locked* to be originally $false$ and sets *m_locked* to *true*. Based on `CBMC`, we extend `Deagle` to support the modeling of more library functions.

Unwinding We employ *bounded model checking* (*BMC*) [3,4,5] to handle loops. If the program contains loops, we determine an *unwinding limit* and unwind the program to a loop-free bounded program:

- If the maximal loop time of the program can be determined through static analysis, e.g.,
$$for\ (i = 0; i < 10; i++)$$
we set the unwinding limit to this maximal loop time;
- If the maximal loop time depends on non-determinism. e.g.,
$$for\ (i = 0; i < n; i++)$$
where n is attained from the function *__VERIFIER_nondet_int*, we report UNKNOWN since such loops cannot be fully unwound.
- Otherwise, we set the unwinding limit to 2.

Formula Generation (extended from `CBMC`) After unwinding, the loop-free program is represented in the *static single assignment* (*SSA*) form, where each thread is a chain of assignments. These assignments can be directly modeled into first-order logic formulas (for ease of solving, we further convert them into propositional logic formulas). Additionally, an assignment may contain global memory access events; we model program orders and read-from orders (please refer to [8] for more information) of these events into the formulas.

Constraint Solving (extended from `MiniSAT`) We develop an ordering consistency theory solver and integrate it into the DPLL(T) framework [8]. For efficiency, we extend `MiniSAT`, an SAT-based solver, to run our theory solver exclusively. Please refer to [8] for the detailed algorithms of our decision procedure.

Witness Generation (adapted from `Yogar-CBMC`) If the back-end solver returns *satisfiable* (i.e., finds a counterexample violating the property), our ordering consistency theory solver reports a sequence (total order) of these events, which can be used for generating the witness of the counterexample.

3 Strengths and Weaknesses

Compared to the traditional method [1] which explicitly converts ordering formulas into propositional formulas, `Deagle` employs a dedicated theory solver to handle ordering formulas, which improves both time and space efficiency. Ignoring some tasks in *goblint-regression* that require unwinding 10000 times, `Deagle` reports TIMEOUT in only 9 tasks and OUT OF MEMORY in only 7 tasks – fewer than most `ConcurrencySafety` competitors.

In most *weaver* tasks (117 out of 169), the number of loop iterations is nondeterministic. As is mentioned in previous section, `Deagle` reports UNKNOWN. Since such tasks are common in real-world programs, we are exploring an approach to dealing with such programs in the future work.

4 Tool Setup and Configuration

The source code of `Deagle 1.3` (the submitted version in `SV-COMP 2022` [2]) is publicly accessible [4]. Please refer to README for more installation instructions. In `SV-COMP 2022`, `Deagle` participates in `ConcurrencySafety` category and only checks property `Unreach-Call` [5]. By setting parameters

$$--32--no-unwinding-assertions--closure$$

one can reproduce `Deagle`'s results of `SV-COMP 2022`.

[4] `Deagle` repository: https://github.com/thufv/Deagle

[5] The benchmark definition of `Deagle`: https://gitlab.com/sosy-lab/sv-comp/bench-defs/-/blob/main/benchmark-defs/deagle.xml

4.1 Parameter Definition

Deagle inherits lots of parameters from CBMC. Due to the page limit, we only describe parameters related to the competition or newly added in Deagle:

* $--32/--64$: sets the width of integers to 32/64.
* $--no-unwinding-assertions$: does not generate *unwinding assertions* into the formula. Assuming a loop is unwound n times, its unwinding assertion asserts the loop condition to be $false$ after n iterations. Since unwinding assertions can lead to false counterexamples, we disable the generation of unwinding assertions.
* $--closure/--icd$ (new in Deagle): uses our proposed approach. Once the parameter $--closure$ is enabled, Deagle employs a transitive closure-based theory solver (recommended). If $--icd$ is enabled, Deagle employs an incremental cycle detection-based solver. In SV-COMP 2022 [2], Deagle solves all tasks with the parameter $--closure$.

5 Software Project

Deagle is developed by Fei He, Zhihang Sun, and Hongyu Fan from the Formal Verification Lab[6] in Tsinghua University. Deagle is licensed under GPLv3. Since Deagle is developed over CBMC and MiniSAT, and reuses some modules from Yogar-CBMC, it also contains copyright of those tools.

6 Acknowledgement

We appreciate SV-COMP hosts for holding the competition and giving advice on participating. We are also grateful to developers, maintainers, and contributors of CBMC, MiniSAT, and Yogar-CBMC, on which Deagle is based.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 141–157. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.5555/2958031.2958083
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
3. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Proceedings of the 36th Annual ACM/IEEE Design Automation Conference. p. 317–320. DAC '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/309847.309942

[6] homepage: https://thufv.github.io/team

4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
5. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (Jul 2001). https://doi.org/10.1023/A:1011276507260
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing. pp. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/3-540-49059-0_14
7. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: CAV (2004). https://doi.org/10.1007/978-3-540-27813-9_14
8. He, F., Sun, Z., Fan, H.: Satisfiability modulo ordering consistency theory for multi-threaded program verification. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 1264–1279. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454108
9. Kroening, D., Tautschnig, M.: CBMC–C bounded model checker. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
10. Yin, L., Dong, W., Liu, W., Li, Y., Wang, J.: Yogar-CBMC: CBMC with scheduling constraint based abstraction refinement. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 422–426. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_25
11. Yin, L., Dong, W., Liu, W., Wang, J.: Scheduling constraint based abstraction refinement for multi-threaded program verification. IEEE Transactions on Software Engineering **PP** (08 2017). https://doi.org/10.1109/TSE.2018.2864122