Cascade (Competition Contribution)

Wei Wang and Clark Barrett

New York University

Abstract. Cascade is a static program analysis tool developed at New York University. It uses bounded model checking to generate verification conditions and checks them using an SMT solver which either produces a proof of correctness or gives a concrete trace showing how an assertion can fail. It supports the majority of standard C features except for floating point. A distinguishing feature of Cascade is that its analysis uses a memory model which divides up memory into several partitions based on alias information.

1 Verification Approach

Bounded model checking (BMC) [4] is an efficient method to detect bugs automatically. The technique constructs a formula that encodes a program up to a user-specified bound. A *memory model* is a crucial part of the encoding in bounded model checking of programs, determining how the contents of and modifications to memory are represented. The most precise model is a *flat* model, which represents memory as a single array of bytes. However, this model typically does not scale well because the solver cannot easily infer which regions are disjoint.

Cascade uses a novel *partition* memory model. The main idea of this model is to split the memory according to the alias information acquired by incorporating a Steensgaard points-to analysis module [8]. This ensures that variables and dynamically allocated regions that may alias end up in the same partition. Each partition is modeled using a separate array. The memory partitioning significantly eases the burden of reasoning about disjointness and thus scales much better than the flat memory model, while the points-to-analysis approach ensures the soundness of modeling type-unsafe behaviors in C.

2 System Architecture

Cascade [9] is implemented in Java. The overall framework is illustrated in Figure 1. The C front-end converts a C program into an abstract syntax tree using a parser built using the xtc parser generator [6]. Both the core module and preprocessing module take the abstract syntax tree as input. In the preprocessing module, the points-to analysis is performed for each function in the C program without function-inlining or loop-unrolling. All the alias groups and the points-to relations among them are discovered here. The core module uses symbolic

execution [2, 3, 7] over the abstract syntax tree to build verification conditions as a SMT formula. Currently, it takes the approach of simple forward execution. The partition memory model is built based on the alias information generated at the preprocessing step. Verification conditions are discharged by an SMT solver. Cascade currently supports both CVC4 [1] and Z3 [5].

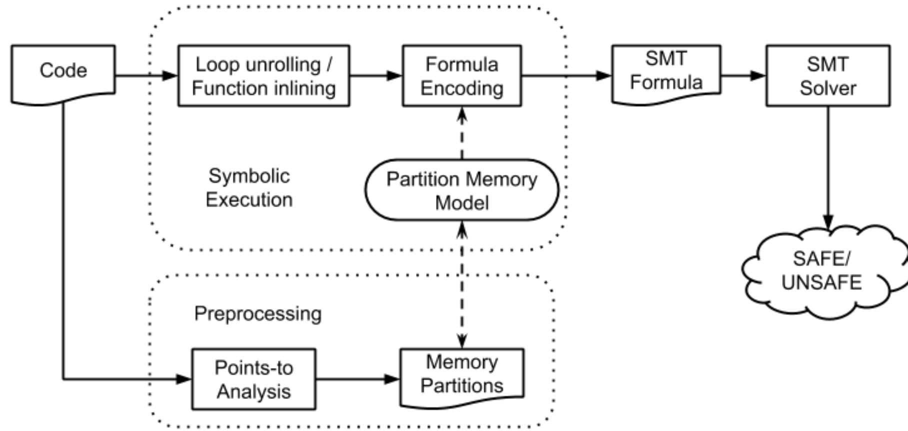

Fig. 1. Cascade framework

3 Strength and Weaknesses of the Approach

Cascade supports arbitrary user assertions, including reachability of labels in the C-code. Furthermore, it can detect bugs related to memory safety, including invalid memory accesses, invalid memory frees and memory leaks. In SV-COMP 2015, these checks are only enabled for the *MemorySafety* category. Cascade relies on loop unrolling and function inlining, and so it may perform poorly if either of these steps are required to be too large. In the competition, Cascade uses successively larger unrolls until a fixed bound of 1024 is reached, or a violation is detected, or a timeout is reached. Note that we set a timeout of 850 seconds. We also use a fixed function-inlining depth of 2. For memory safety checking, we use a different set of parameters: the maximum unroll is 200 and the inline depth is 5. If no error is found or the `ERROR` label cannot be reached within the maximum bounds, Cascade will report `SAFE`. Otherwise, it will report `UNSAFE` and the witness will be dumped in the GraphML format.

4 Tool Setup and Configuration

The version of Cascade submitted to SV-COMP 2015 can be downloaded at:

`http://cascade.cims.nyu.edu/bin/sv-comp-2015-4113-cvc4-patch.tar.gz`

This version uses CVC4 as the back-end solver. Cascade requires JVM version 1.7.0. The archive unzips to a directory called `sv-comp-2015-4113-patch` which contains a script called `run_cascade_bmc`. The script should be run from the `sv-comp-2015-4113-patch` directory as follows:

```
run_cascade_bmc -trace <c-benchmark>
```

where `c-benchmark` is the name of the C file to be analyzed. The results are printed on stdout and should be interpreted as follows:

- if the last line printed is `UNSAFE`, this should be interpreted as `FALSE`;
- if the last line printed is `UNSAFE:p_<prop>` this should be intepreted as `FALSE(<prop>);`
- otherwise, if the last *word* printed is `SAFE`, this should be interpreted as `TRUE`;
- any other result should be interpreted as `UNKNOWN`.

For results that correspond to `FALSE`, a witness is dumped to the file:

```
out/<benchmark-name>/witness.graphml.
```

where `<benchmark-name>` is the filename of the C benchmark that was checked without the path prefix.

In the competition, Cascade will participate in the following categories: *Bit Vectors*, *Control Flow and Integer Variables*, *Heap Manipulation*, and *Memory Safety*. We will not participate in the others for various reasons including lack of support for function pointers and concurrency.

References

1. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. *CAV '11*, 6806:171–177, 2011.
2. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. *Proceedings of Design Automation Conference (DAC'99)*, 317:226–320, 1999.
3. D. Brand and W. H. Joyner. Verification of protocols using symbolic execution. *Comput. Networks*, 2:351, 1978.
4. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *Proc. TACAS 2004*, 2988:168–176, 2004.
5. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS*, pages 337–340, 2008.
6. R. Grimm. Rats!, a parser generator supporting extensible syntax. 2009.
7. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 385:226–394, 1976.
8. B. Steensgaard. Points-to analysis in almost linear time. *ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.
9. W. Wang, C. Barrett, and T. Wies. Cascade 2.0. *VMCAI '14*, 2014.