Cascade: C Assertion Checker and Deductive Engine (Tool Submission)

Nikhil Sethi and Clark Barrett

New York University, New York, NY, USA
{nsethi,barrett}@cs.nyu.edu

Abstract. We present a tool, called Cascade, to check assertions in C programs as part of a multi-stage verification strategy. Cascade takes as input a C program and a *control file* (the output of an earlier stage) that specifies one or more assertions to be checked together with (optionally) some restrictions on program behaviors. For each assertion, Cascade produces either a concrete trace violating the assertion or a deduction (proof) that the assertion cannot be violated.

1 Introduction

Software verification is an active area of research [2, 3, 5, 6, 9, 10]. Tools have been developed which can find bugs in real applications with large code bases. However, in order to analyze large programs, these tools often make approximations. As a result, some of the errors reported by such tools can be false.

A promising alternative approach is the idea of two-stage verification [2, 7, 8]. In two-stage verification, a light-weight analysis capable of scaling to large programs is run first to identify potential bugs. This is followed by a more detailed analysis of the potential errors identified in the first stage. Cascade provides a generic back-end for two-stage verification of C programs which can be easily integrated with any initial stage. Cascade can handle most C constructs including loops, functions (including recursive functions), structs, pointers, and dynamic memory allocation.

2 System Description

Cascade consists of about 6000 lines of C++ code. Its overall design is shown in Fig. 1. The core module takes as input an abstract syntax tree representing a C program and a *control file* that specifies one or more potential errors to be checked. The core module uses symbolic simulation over the abstract syntax tree to build verification conditions corresponding to the assertions specified in the control file. The semantics of C statements are hard-coded into the translation rules that the core module uses to convert C statements into logic formulas. Cascade uses a bounded model-checking approach to handle loops (and recursive functions). Loops are unrolled a fixed number of times (this number can be specified by the user). Cascade models all pointers and addresses in the heap precisely. The *data* stored in memory is represented abstractly as integers.

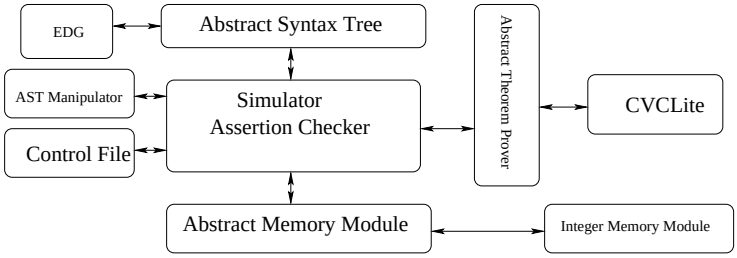

Fig. 1. Cascade: System design

2.1 Abstract Interfaces

Cascade is designed to be easily customizable. Major components are hidden behind abstract interfaces. This makes it easy to provide and experiment with a variety of configurations based on the same basic architecture. The core module depends on implementations of three generic abstract interfaces: an abstract syntax tree, a memory module, and a theorem prover.

Abstract Syntax Tree. Cascade has a simple internal representation of programs as an abstract syntax tree. All operations are done on this internal representation, completely separating it from the front end which is responsible for building the abstract syntax tree. Cascade currently has an implementation using EDG [4], an industrial-strength parser for C programs, as a front end to create the abstract syntax tree.

Abstract Theorem Prover. Cascade uses an abstract theorem proving interface. The interface provides an abstract `ExprNode` object which corresponds to logical expressions in the underlying theorem prover. It also specifies some standard operations on `ExprNodes` like arithmetic operations, Boolean operations and array operations. Any theorem prover which can support these operations can easily be used with Cascade. An unsupported operation can be set to return *unknown*. Cascade currently uses CVC Lite[1] as its theorem prover. CVC Lite can produce proofs and concrete counter-examples. An additional advantage of using CVC Lite is our in-house expertise on using and modifying the theorem prover.

Abstract Memory Module. All memory operations during simulation are handled by an abstract interface modeling heap memory. Memory is a mapping from addresses to values where both of them are `ExprNode` objects. Functions like allocate, deallocate, read and write are supported. The memory module also provides a `check_valid_address` function which checks if a given address is valid or not. This function can be used in assertions. The current implementation of the abstract memory module uses an array of integers to model memory. We expect to provide a more precise model of the data in memory using bit-vectors (which are supported in the latest version of CVC Lite) in the near future.

2.2 Core Module

Simulator. The simulator integrates the various modules. It symbolically simulates the program, building expressions using the abstract interface to the theorem prover, and then checking the expressions corresponding to the assertions specified in the control file. An interface to the simulator is also exported, enabling Cascade to be used as a library that can be integrated with other tools.

AST Manipulator. The AST manipulator module has various functions which can modify the AST. For example, unrolling of the loops is handled by this module. This module also interfaces with the control file and integrates the restrictions on execution paths and variables with the AST.

2.3 The Control File

Execution of the tool is guided by a **Control File**. A control file specifies the assertion(s) to be checked. In addition, a control file can be used to constrain the search for a violating trace by restricting the program paths to be explored or giving constraints on program variables. The control file allows important information about feasible violations (perhaps gathered by an earlier stage) to be communicated to Cascade.

The control file has a simple XML format. It begins with `SourceFile` sections which give the paths to C source files. It then has one or more `Run` sections, each defining a constrained run of the program. Each run starts with a single `StartPosition` and ends with a single `EndPosition` section. These give respectively the start point and end point of the simulation to be run. A `Run` may optionally specify one or more `WayPoint` sections. A `WayPoint` indicates that Cascade should consider only those program paths which pass through the `WayPoint`. Each position (start, end, or waypoint) can also include a command. Commands include: `cascade_assume`, which takes a Boolean C expression and adds it as an assumption to the theorem prover; `cascade_check`, which takes a Boolean C expression and checks whether it is valid at the given position; and `cascade_check_valid_address`, which takes a C expression as its argument and checks if the address represented by the expression is a valid address in memory.

3 An Example

Table 1 gives a small C function which has a `NULL` pointer access if its argument is negative. Suppose that a suitable first-stage tool [6, 7, 10] has flagged Line 9 as a potential error. With no further information, Cascade finds a violating trace in which the argument is negative. However, suppose the first-stage tool knows from its analysis that the function f is only called with a positive argument. Using the control file, the first-stage can constrain the search to only those cases when $a > 0$. In this case, Cascade can verify that the assertion cannot be violated. The code and control file for this example are shown below.

Table 1. Control file example

```
1  int* f(int a) {
2    int *p, *x, *y;
3    x = (int*) malloc(sizeof(int));
4    y = NULL;
5    if(a>=0)
6       p = x;
7    else
8       p = y;
9    *p = 5;
10   return p;
11 }
```

```
<ControlFile>
    <SourceFile>
      <Name>~/ex/f.c</Name><FileId>1</FileId>
    </SourceFile>
    <Run><StartPosition><Position>
      <FileId>1</FileId><LineNum>1</LineNum>
      </Position><Command>
        <CascadeFunction>cascade_assume
        </CascadeFunction>
        <Argument>a>0</Argument>
      </Command></StartPosition>
      <EndPosition><Position>
        <FileId>1</FileId><LineNum>9</LineNum>
      </Position><Command>
        <CascadeFunction>
          cascade_check_valid_address
        </CascadeFunction>
        <Argument>p</Argument>
      </Command></EndPosition></Run>
</ControlFile>
```

4 Conclusion

Cascade has been successfully run on programs of up to a few hundred lines of code. For a 400 line example, without any constraints in the control file, the run-time on a P4 2GHz is less than 1 minute. We expect that with suitably constrained control files, Cascade will scale to much larger code bases. Although it is still under development, we hope it will be of use and interest to a broader community. In addition, we hope to receive feedback and suggestions for further improvement. For further information on Cascade, including downloads, examples and documentation, see `http://www.cs.nyu.edu/acsys/cascade/`.

References

1. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of CAV*, pages 515–518, July 2004.
2. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with blast. In *FASE*, pages 2–18, 2005.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of PLDI*, pages 196–207, 2003.
4. Edison design group. `http://www.edg.com`.
5. S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of PLDI*, pages 69–82, 2002.
6. G. J. Holzmann. Static source code checking for user-defined properties. In *Proc. IDPT 2002*, Pasadena, CA, USA, 2002.
7. Orion. `http://cm.bell-labs.com/cm/cs/what/orion/index.html`.
8. X. Rival. Understanding the origin of alarms in ASTRÉE. In *SAS*, volume 3672 of *LNCS*, pages 303–319, London (UK), Sept. 2005. Springer.
9. Y. Xie and A. Aiken. Saturn: A sat-based tool for bug detection. In *CAV*, pages 139–143, 2005.
10. Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of ESEC/FSE*, pages 327–336, 2003.