PredatorHP and SV-COMP 2017
(Competition Contribution)*

Michal Kotoun, Petr Peringer, Veronika Šoková, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. This paper describes shortly the PredatorHP (Predator Hunting Party) analyzer and its participation in the SV-COMP 2017 software verification competition. The paper starts by a brief sketch of the Predator shape analyzer on which PredatorHP is built, using multiple, concurrently running, specialised instances of Predator.

1 Verification Approach

Predator Hunting Party (PredatorHP) uses the Predator shape analyzer, and so we first give a brief overview of Predator. Next, we discuss how Predator is used in the concurrent setting of PredatorHP.

1.1 The Predator Shape Analyzer

Predator aims at *sound* shape analysis of sequential, non-recursive C programs that use various kinds of lists implemented using low-level C pointer statements. Predator can soundly deal with various forms of pointer arithmetics, address alignment, block operations, memory contents reinterpretation, etc.

The shape analysis implemented in Predator is a form of *abstract interpretation* which uses a domain of the so-called *symbolic memory graphs* (SMGs) [1]. SMGs are oriented graphs with two main kinds of nodes and two main kinds of edges. Nodes can be divided into *objects* and *values*. Objects are further divided into *regions* (representing concrete blocks of memory allocated on the stack, on the heap, or statically) and singly- or doubly-linked *list segments*, which represent in an abstract way uninterrupted sequences of singly- or doubly-linked regions. Edges can be divided into *has-value* and *points-to* edges. The former represent values stored in allocated memory (which are either pointers or other kinds of data), the latter represent targets of pointer values.

Both nodes and edges are annotated by a number of *labels* that carry information such as the size of objects, offsets at which values are stored in objects, offsets with which pointers point to target objects, the type of values, offsets at which linking fields of lists are stored, the nesting level of objects (to be able to represent nested lists), or a constraint on the number of linked regions that a list segment represents. In particular, a list segment can either represent n or more regions for $n \geq 0$, or 0 or 1 regions. Further, SMGs can also contain *optional regions* where a pointer to such a region either points to some allocated memory or to `NULL`. Sizes of blocks and offsets can have the form of *intervals* with constant bounds which allows Predator to deal with operations

* The work was supported by the Czech Science Foundation project 14-11384S.

such as address alignment. A special kind of edges are then *disequality edges* allowing one to express that two values are for sure different (while equality of objects is expressed by representing these objects by a single node of an SMG).

Symbolic execution of C statements on SMGs uses a concept of *reinterpretation* that is able to synthesize values of previously not explicitly written fields from the known values of other fields. Currently, this concept is instantiated for dealing with blocks of nullified memory, which is quite needed for analyzing low-level programs. Another key operation on SMGs is the *join operation* that is implemented via a synchronous graph traversal of the two SMGs to be joint. The join operation is used not only to reduce the number of SMGs to deal with but also as a basis of abstraction and entailment checking. Predator uses *function summaries* to facilitate inter-procedural analysis. The support of *arithmetic* in Predator is such that Predator deals with integers exactly up to some bound (32 in SV-COMP'17) and then replaces them by an unknown value.

Compared with SV-COMP'16 [3], not many changes were done in the Predator analyzer itself. We have just resolved several minor issues by, e.g., replacing error messages produced when performing so-far unsupported operations over bitfields by producing the "unknown" verdict. Further, some needed changes in the witness format were implemented.

1.2 Predator Hunting Party

For SV-COMP'17, we have decided to use the same concept of PredatorHP as in SV-COMP'16. PredatorHP is implemented as a Python script which runs several instances of Predator in parallel and composes the results they produce into the final verification verdict. In particular, PredatorHP starts four Predators: One of them is the original Predator that soundly overapproximates the behaviour of the input program — we denote it as the *Predator verifier* below. Apart from that, three further Predators are started which are modified as follows: Their join operator is reduced to joining SMGs equal up to isomorphism and they use no list abstraction. Two of them use a bounded depth-first search to traverse the state space. They use bounds of 200 and 900 GIMPLE instructions, and so we call them as *Predator DFS hunters*. Third of them — *Predator BFS hunter* use an unlimited breadth-first search to traverse the state space.

If the Predator verifier claims a program correct, so does PredatorHP, it create correctness witness[1] and it kills all other Predators. If the Predator verifier claims a program incorrect, its verdict is ignored since it can be a false alarm (and, moreover, it is highly non-trivial to check whether it is false or not due to the involved use of list abstractions and joins). If one of the Predator DFS hunters finds an error, PredatorHP kills all other Predators and claims the program incorrect, using the trace provided by the DFS hunter who found the error as a violation witness. One hunter searches quickly for bugs with very short witnesses and one than searches for longer but still not very long witnesses.

If a DFS hunter claims a program correct, its verdict is ignored since it may be unsound. If a BFS hunter manages to find an error within the SV-COMP'17 time budget, PredatorHP claims the program incorrect (note that without a time limit, the BFS hunter

[1] Format for violation and correctness witnesses: `https://github.com/sosy-lab/sv-witnesses/`

is guaranteed to find every error). If the BFS hunter finishes and does not find an error, the program is claimed correct. Otherwise, the verdict "unknown" is obtained.

2 Strengths and Weaknesses

The main strength of PredatorHP is that—unlike various bounded model checkers—it treats unbounded heap manipulation in a *sound* way. At the same time, it is also quite *efficient*, and the use of various concurrently running Predator hunters greatly decreases chances of producing *false alarms* (there do not arise any due to heap manipulation, the remaining ones are due to imprecise treatment of other data types).

The main weakness of PredatorHP and also of Predator itself is its weak treatment of non-pointer data. Due to this, Predator participates in the heap data structures category only. Within this category, a weakness of Predator is that it is specialized in dealing with lists, and hence it does not handle structures such as trees or skip-lists (that is, it handles them very well in a bounded way, but our aim is to stick with sound verification).

3 Tool Setup and Configuration

The source code of PredatorHP used in SV-COMP'17 is freely available on the Internet[2]. The file `README-SVCOMP-2017` shipped with the source code describes how to build the tool. To run it, the script `predatorHP.py` can be invoked. The script takes a verification task file as a single positional argument. Paths to both the property file and the desired witness file are accepted via long options. The verification outcome is printed to the standard output. The script does not impose any resource limits other than terminating its child processes when they are no longer needed. More information about the setting of PredatorHP used in the competition can be found here: `http://sv-comp.sosy-lab.org/2017/systems.php`.

4 Software Architecture, Project, and Contributors

Predator is implemented in C++ with a use of Boost libraries as a GCC plug-in based on the Code Listener framework [2]. PredatorHP is implemented as a Python script. Predator is an open source software project distributed under the GNU General Public License version 3. The main author of Predator is Kamil Dudka. Besides him and the PredatorHP team, Petr Müller, and numerous other people contributed to Predator.

References

1. K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *Proc. of SAS'13*, *LNCS* 7935, pages 214–237, Springer, 2013.
2. K. Dudka, P. Peringer, and T. Vojnar. An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proc. of EUROCAST'11*, *LNCS* 6927, pages 527–534, 2012.

[2] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator-hp`

3. M. Kotoun, P. Peringer, V. Šoková, and T. Vojnar. Optimized PredatorHP and the SV-COMP Heap and Memory Safety Benchmark (Competition Contribution). In *Proc. of TACAS 2016*, *LNCS* 9636, pages 942–945, 2016.