Predator: A Shape Analyzer Based on Symbolic Memory Graphs (Competition Contribution)[⋆]

Kamil Dudka, Petr Peringer, and Tomáš Vojnar

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

Abstract. Predator is a shape analyzer that uses the abstract domain of symbolic memory graphs in order to support various forms of low-level memory manipulation commonly used in optimized C code. This paper briefly describes the verification approach taken by Predator and its strengths and weaknesses revealed during its participation in the Software Verification Competition (SV-COMP'14).

1 Verification Approach

Predator is a shape analyzer that uses the abstract domain of *symbolic memory graphs (SMGs)* in order to support various forms of low-level memory manipulation commonly used in optimized C code. Compared to separation logic-based works [1], which our work is inspired by, SMGs allow one to easily apply various graph-based algorithms to efficiently manipulate with the low-level memory representation.

The formal definition of SMGs can be found in [2] together with algorithms of all the operations needed for use of SMGs in a fully automatic shape analysis. This is in particular the case of a specialised unary abstraction operator and a binary join operator that aid termination of the SMG-based shape analysis. The join operator is based on an algorithm that simultaneously traverses a pair of input SMGs and merges their corresponding nodes. The core of the join algorithm is also used by the algorithm implementing the abstraction operator to merge pairs of neighbouring nodes, together with their sub-SMGs (describing the data structures nested below them), into a single list segment. For checking entailment of SMGs, Predator again reuses the join algorithm (extended to compare generality of the SMGs being joined).

Predator requires all external functions to be properly modelled wrt. memory safety in order to exclude any side effects that could possibly break soundness of the analysis. Our distribution of Predator includes models of memory allocation functions (like `malloc` or `free`) and selected memory manipulating functions (`memset`, `memcpy`, `memmove`, etc.).

Since SV-COMP'13, the core algorithms of shape analysis were reimplemented in order to match their description presented in [2]. Consequently, the current implementation is much easier to follow, but at the same time also faster and more precise (as witnessed by the results of SV-COMP'14).

[⋆] This work was supported by the Czech Science Foundation project 14-11384S and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

2 Software Architecture

Predator is implemented as a GCC (GNU Compiler Collection) plug-in, which makes the tool easy to use without a need to manually preprocess the source code. GCC as an industrial-strength compiler takes care of parsing the C code into an intermediate representation (known as GIMPLE). The input code is symbolically executed by Predator using the algorithms proposed in [2] with the aim to precisely interpret various low-level memory operations (such as pointer arithmetic, valid use of pointers with invalid targets, operations with memory blocks, or reinterpretation of the memory contents). Predator is written in C++ and requires Boost libraries, mainly to enable using legacy compilers for building it. The Predator GCC plug-in can be loaded into any GCC with a plug-in support up to GCC 4.8.2 (which was the latest release in 2013).

Compared to SV-COMP'13, Predator uses an improved algorithm for live variable analysis (based on a points-to analysis). The improved live variable analysis makes the shape analysis run five times faster in certain cases (e.g. the Merge-Sort algorithm case study from [2]).

3 Strengths and Weaknesses

The main strength of Predator is its byte-precise representation of reachable memory configurations, which makes it possible to successfully verify certain low-level pointer-intensive programs in the *MemorySafety* and *HeapManipulation* categories. The key design principle of Predator is soundness, which was again confirmed by reaching zero false negatives on the whole benchmark of SV-COMP'14. On the other hand, Predator does not check spuriousness of possible counterexamples for now, which caused numerous false positives (and consequently a significant loss of score). Since SV-COMP'13, the *MemorySafety* category has been extended by case studies that cause problems to Predator either by operating on data structures not covered by the current abstraction algorithm (trees and skip lists), or by the requirement to track non-pointer data along with the shapes of data structures (for example, tracking the length of lists). Compared to the SV-COMP'13 version, Predator now finally achieved the full score in the *ProductLines* subcategory. Moreover, the correct results were now delivered five times faster than the partially correct results in this (sub)category last year.

Results in the *ControlFlowInteger BitVectors* categories still suffer from a high ratio of false positives caused mainly by a too coarse analysis of integers. Due to undefined external functions, Predator was not able to analyze many test cases in the *DeviceDrivers64* and *Concurrency* categories.

4 Tool Setup and Configuration

The source code of the Predator release[1] used in the competition can be downloaded from the project web page. The file `README-SVCOMP-2014` included in the archive

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/download/predator-2013-10-30-d1bd405.tar.gz`

describes how to build Predator from source code and how to apply the tool on the competition benchmarks. After successfully building the tool from sources, the `sl_build` directory contains a script named `check-property.sh`, which needs to be invoked once for each input program. Besides the name of the input program, the script requires a mandatory option `--propertyfile` specifying the property to be verified. Compiler flags needed to compile the input program with GCC must be specified after the file name of the input program. For programs relying on a particular target architecture (such as preprocessed C sources), it is important to use the `-m32` or `-m64` compiler flags to specify the architecture. The script also provides a voluntary option `--trace` that allows one to write the error trace to a file. The verification result is printed to the standard output on success. Otherwise, the verification outcome should be treated as `UNKNOWN`. The script does not check for exceeding any resource limits on its own.

Although we use a global configuration of Predator for all categories, the tool provides many useful compile-time options via the `sl/config.h` configuration file. The default configuration is tweaked to obtain good overall results in both the competition benchmark and Predator's regression test-suite. The configuration can be further tweaked to improve the results in a particular category, however, at the cost of loosing some points in other categories.

5 Software Project and Contributors

Predator is an open source software project developed at Brno University of Technology (BUT) and distributed under the GNU General Public License version 3, which allows Predator to be used for both commercial and non-commercial purposes. There is no binary distribution of Predator, but it can be easily built from sources on any up to date distribution of Linux. The interaction with the compiler is facilitated by the Code Listener infrastructure [3], which is shared with Forester (a shape analyser based on forest automata [4]), including a suite of regression tests. Both Code Listener and Forester are projects also developed at BUT. Besides our development teams, we have numerous external contributors listed in the `docs/THANKS` file inside the distribution of Predator. Collaboration on further development of Predator is welcome.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. CAV'07*, *LNCS* 4590, Springer, 2007.
2. 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.
3. 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, 2013.
4. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV'11*, LNCS 6806, Springer, 2011.