Predator: A Verification Tool for Programs with Dynamic Linked Data Structures⋆
(Competition Contribution)

Kamil Dudka, Petr Müller, Petr Peringer, and Tomáš Vojnar

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

Abstract. Predator is a tool for automated formal verification of sequential C programs with dynamic linked data structures. It is in principle based on separation logic, but uses a graph-based heap representation. This paper first provides a brief overview of Predator and then discusses experience with its participation in the Software Verification Competition of TACAS'12.

1 Introduction

Predator is a tool for automated formal verification of sequential C programs with dynamic linked data structures. Currently, it supports verification of various linked list variants, including nested, cyclic, and/or shared lists, possibly using limited pointer arithmetics to navigate through list nodes as is usual in real-life implementations of list manipulating programs. Predator implicitly detects various memory-related errors and can also check for reachability of error labels, which made its participation in the TACAS'12 Software Verification Competition (SV-COMP'12) possible. However, most of the capabilities of Predator to detect memory-specific errors could not be applied in the competition. Predator is publicly available[1] as open-source under GPLv3.

This paper provides a brief overview of Predator's design principles and capabilities, and then discusses experiments with Predator on the benchmarks of SV-COMP'12. More details about Predator can be found in the tool paper [1].

2 Overview of Predator

Predator is conceptually based on *separation logic* with *higher-order inductive predicates*. It encodes infinite sets of heaps in a finite symbolic way using a *graph-based representation* of separation logic formulae. There are two kinds of nodes in the graphs: (1) possibly nested *objects* corresponding to statically and automatically allocated program variables, dynamically allocated storage, list segments, etc. and (2) *values* of the objects, e.g., addresses of objects and the special values `undefined`, `deleted`, and `null` in the case of pointers and function pointers. This allows for dealing with pointers to any variable (not only dynamically allocated) and detection of some classes of memory-related bugs (stack smashing, buffer overrun, and the like). Predator also implicitly detects other memory-related errors like memory leaks, invalid dereferences, and double frees. Predator uses a specialised *join operator*, applicable both on entire symbolic heaps as well as on their parts. The latter functionality serves for discovering

---

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

new list predicates used for a subsequent abstraction (summarisation) of list segments. The join algorithm is also used for checking entailment of symbolic heaps (by checking that the join of two symbolic heaps produces one of them). Hence, Predator does not use any off-the-shelf decision procedures.

The main goal of Predator is to verify real system code in a fully automated way. Since real-life implementations of lists often use limited *pointer arithmetics* (e.g., in Linux, the list header structure is embedded at any place in list nodes, and pointer arithmetics is used to move within the list nodes), Predator is capable of handling typical patterns of such low-level programming techniques. For this, *offsets* of sub-objects within their encapsulating objects are tracked.

Predator is implemented as a *GCC plug-in*[2], which brings significant advantages. For the many real-life programs whose production versions are built using GCC, the analysis is performed on the same program representation as the one used for producing the actual binary. GCC has also a very large coverage of what it can parse (standard C and GNU extensions), while other tools, like CIL, used by some other analysers, usually implement only a subset of the C language standard. Further, using GCC allows an easier integration with other commonly used development tools. In particular, Predator presents errors and warnings in the standard GCC format, which many development tools can handle by default.

Predator is written in C++ and uses Boost libraries. The only dependencies that need to be installed are GCC 4.4.6+ with C++ support and CMake. It is recommended to build Predator against a local build of GCC. This can be done in several steps, which are described in the README file[3] of the Predator's distribution. The local build of GCC is fully automated and the whole process takes up to 10 minutes assuming a fast enough Internet connection (needed for downloading GCC sources). The current version of Predator runs on Linux, but the code of the analyser itself is architecture-independent.

3 Experience with Predator

Predator is distributed with a collection of more than 200 test cases. The test cases include real-world code snippets as well as code focused on corner cases in the use of the dynamic linked data structures. This collection also serves as our internal benchmark for measuring the performance of the tool. The errors sought in these benchmarks are typical memory manipulation errors (memory leakage, double free, etc.) for which no user-provided specification is needed. More information on our test cases can be found in [1]. Four of our test cases were extended by explicit checks of shape properties and sent as a contribution to the SV-COMP'12 benchmarks.

Below, we summarise our experience with Predator from the training phase of SV-COMP'12, describe the results we reached on the competition benchmarks, and discuss problems encountered on particular test cases. We refrain from stating precise quantitative data about our results, which is a part of the presentation of SV-COMP'12 itself.

During the experiments with the training set of test cases of SV-COMP'12, we encountered problems resulting from Predator's implicit detection of memory-related errors (such as invalid dereferences or buffer overruns), which Predator never ignores.

---

[2] Predator uses the low-level GIMPLE representation of the GCC's intermediate code.

[3] Instructions specific for the Software Verification Competition held at TACAS'12 are located in the file named README-sv-comp-TACAS-2012 available in the distribution of Predator.

Some of the test cases in the benchmark were problematic from this point of view since they assumed idealised memory models. These test cases caused Predator to terminate prematurely and report memory errors unrelated to the reachability of the given error label. Hence, we needed to create a special layer on top of Predator that distinguishes between errors defined by the competition rules (where UNSAFE means that an error label is reachable) and errors caused by using memory in a wrong way.

From the SV-COMP'12 categories, we focused on those that Predator is designed to verify, especially on the Dynamic Data Structures category. This category contained test cases contributed by the Predator project itself (*heap-manipulation*) and test cases taken from the web page of the BLAST 3.0 project (*list-properties*). In this category, Predator succeeded in all but two test cases, which contained lists with alternating small integral numbers in their nodes. The current version of Predator cannot represent numbers in an abstract way (as, e.g., integral ranges), which prevented the list segment abstraction from being applied and consequently the analysis failed to terminate.

Verifying Linux drivers is one of the major goals for Predator, so we expected good results in this category too. On the *ldv-regression* benchmark, we ended up with the highest possible score (at least in our home environment). For that to happen, it was necessary to improve some test cases to make them allocate the memory they use and to write a few dummy models of external functions, but these changes were accepted by the competition organisers. Unfortunately, Predator was not successful in the *ddv-machzwd* benchmark due to the lack of abstraction over integral values as mentioned above.

Across all benchmark categories, Predator never returned SAFE for a test case declared UNSAFE, which confirms that the analysis done by Predator is sound. On the *pthread* benchmark, Predator instantly returned UNKNOWN for all test cases because of an unhandled call to pthread_create(). As Predator does currently not aim at the analysis of concurrent programs, this was an expected response. In the *locks* benchmark, Predator could solve all of the test cases, however, given time and space larger than allowed by the rules of SV-COMP'12. In the given limits, Predator managed to analyse only a few of these test cases. This is due to the analysis done by Predator is quite inefficient for such kind of programs since no (refinable) abstraction of non-pointer data is currently supported by Predator. Such data is either tracked precisely or completely discarded. Predator also succeeded on several test cases from the *systemc* benchmark, which were proven SAFE. Like in the *ddv-machzwd* benchmark, Predator lost many points here because of the lack of abstraction over integral values.

4 Conclusions and Future Work

We have briefly presented Predator and its participation in SV-COMP'12. Predator is regularly updated and enhanced. We plan to add support for further kinds of dynamic data structures (like trees), improve the support for non-pointer data, provide support for analysing C++ code, and possibly add techniques allowing one to analyse incomplete programs (e.g., using bi-abduction). SV-COMP'12 provides many interesting test cases, which represent a good motivation for further development of Predator.

References

1. K. Dudka, P. Peringer, and T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In *Proc. of CAV'11*, *LNCS* 6806, Springer, 2011, pp. 372–378.