BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Angie: A Framework for Static Analysis over Symbolic Memory Graphs and Beyond

Michal Kotoun*, Michal Charvát**

Abstract
Creating a software verification tool is a complex task: Source parsing, instruction representation, value abstraction, user interface, ... the analysis itself. Therefore, it was decided to create a static analysis framework to free analysis implementers of all the unnecessary wheel reinventing.
We propose a general design of the framework with a primary focus on usability, having a work-in-progress implementation of the framework and model analysis based on Symbolic Memory Graphs. The programming language of choice is C++ with LLVM serving as the front-end for parsing the source code of analysed program.
We aim to surpass Predator implementation of Symbolic Memory Graphs in the means of abstraction precision and to prove usefulness of this framework, with hopes that it will attract more people to work on static analysis and verification.

Keywords: Static Analysis Framework — Verification — C — C++ — LLVM — Predator — Symbolic Memory Graphs — Abstract Interpretation

Supplementary Material: Code Repository

*xkotou04@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*
**xcharv16@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

As the amount of software critically influencing our lives gradually increases, so rises the importance of different methods for checking the software for flaws. There are several well-known historical examples of what a software error can cause from the areas of space missions or pharmacy, and companies in such fields are still searching for the ultimate verification and bug-hunting tools. In this paper, out of the broad range of approaches that such tools can be based on, we are particularly interested in static analysis methods.

A static analyser for languages like C is a complex piece of software. It includes much more than just the analysis algorithms themselves: it must be able to compile all the details of the source language, provide a viable abstraction/simplification over its structures and commands into some intermediate representation, provide infrastructure for combining different analyses, and be able to report results of the analysis.

Our work is inspired by a verifier for low-level C programs with dynamic linked data structures called *Predator* [1]:

- It is designed as a compiler plug-in.
- It is built on top of the Code Listener framework [2] — an interface to access an intermediate representation of program from the GNU GCC (default) or *LLVM* clang [1]
- Its verification loop is based on abstract interpretation instantiated by *Symbolic Memory Graphs* [2]

Predator is quite efficient and can handle many complex program constructions. It is a multiple-time winner [3, 4, 5] of the *heap memory* and *memory safety* categories in SV-COMP[3] despite not handling recur-

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/
[2] http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/
[3] International Competition on Software Verification, https://sv-comp.sosy-lab.org

sion, modular programs and integers above specified limit.

We – as the members of VeriFIT group – intends to build on *Predator*'s success and push its usability border even further. However, this is not so easy:

First, the architecture of *Predator* was written to a large degree by a single developer who did not think much of later extensions of the tool and who left the team, and hence it is difficult to even understand all details of the tool. Moreover, *Predator* (and also the related tool *Forester* that shares with Predator the *Code Listener* infrastructure), is very optimized and any changes to it pose a great challenge (and use to pose it even to its original author).

Of course, one can start from the scratch – and do this every time a new analysis is needed. However, taking into account what has already been said above about what all needs to be included into a reasonable tool, the developer of the new analysis will be forced to reinvent the wheel by creating a waste number of base modules, like e.g. front-ends, intermediate representation, etc. Therefore, it was decided that a complete re-implementation of Predator is needed, but it should be done in such a way that it should be easy for new developers to join its development and/or implement a new analysis within the created infrastructure.

Therefore we need software that:

- simplifies creating new analyses as much as possible,
- handles abstracting instructions from source language into some reasonable intermediate form,
- allows for composing analyses,
- implements common parts of most analyses as built-in components.

This fulfils the definition of a *framework*. In this paper, we describe our work on such a framework that we call *Angie* and we hope that it will make implementation of new program analysis for researchers smooth and simple. The initial scope for supported input languages should be C with possible extension for C++ later.

There are of course some related works devoted to development of similar frameworks. Let us name the most well-known of them:

- *CPAchecker*[4] is a platform for software verification written in Java and based on the idea of Configurable Program Analysis (CPA). It allows for composition of analysis and one of the supported analyses is even a simplified version of the analysis implemented in Predator (so far without a support for abstraction). Unfortunately, CPAchecker forces developers of the analyses to be used within it to accept the approach of CPA which may sometimes be restricting.
- *Ultimate*[5] is also written in Java, uses a dialect of *Boogie*[6] for intermediate representation, and concentrates on analyses implemented in an automata-theoretic way. The latter is again somewhat restricting.
- *Frama-C* [7] is one of the most known *plug-in* based platform for C analysers, it operates on *CIL*—the *C Intermediate Language*[8] and is written in OCaml as many others research verification tools. Here, the use of CIL is not much aligned with the latest development in the world of major compilers such as those of LLVM[9] family.

In our infrastructure, we would like to rely on a major compiler infrastructure, and we would like to be as little restricting in terms of the kind of analyses supported as possible.

We will explain later that we decided to use *LLVM* for the intermediate representation of programs, so we would like to mention two further recent tools that also use the *LLVM* tool-chain as their front-end and also compete in *SV-COMP*:

- *Symbiotic* [10] is a verifier based on symbolic execution written in C++.
- *DiVinE* [11] is also written in C++, focuses on verification of concurrent programs, features a virtual machine interpreting the *LLVM bitcode*, debugger displaying the program in *LLVM assembly language* and others.

None of these tools, however, can be viewed as an infrastructure.

We designed Angie to be modular and not restrictive from the point of view of the way the analyses should be implemented in it. It should allow combined analyses to be implemented in it. However, we do not intend to provide any super-generic means for communication among the analyses implemented in the

[4] https://cpachecker.sosy-lab.org/

[5] http://monteverdi.informatik.uni-freiburg.de/ultimate/

[6] https://github.com/boogie-org/boogie

[7] https://frama-c.com/

[8] https://people.eecs.berkeley.edu/~necula/cil/

[9] http://llvm.org/

[10] https://github.com/staticafi/symbiotic/

[11] https://divine.fi.muni.cz/

framework. We let it on the developers of the analyses to combine them. Intuitively, a combined analysis should be added as a new analysis, but creating it should be relatively easy as the framework should ease creation of any analysis in general by providing the front-end, intermediate representation, as well as some light-weight communication means among the analysis (that can but need not be used by developers of combined analyses).

We chose *LLVM* to be the fronted since the *LLVM* tool-chain has a stable development and provides us with a way to support many input languages. The first and model analysis we are implementing in Angie is a shape analysis based on the *Symbolic Memory Graphs* originally introduced in Predator, to which we are making some improvements allowing for a more precise abstraction.

We have chosen C++ to be the implementation language because it is one of the most used ones and because *LLVM* itself is implemented in it. We are aware of the fact, that C++ might not be the easiest language to use when implementing a new analysis, but we hope this choice will prove to be acceptable as students and young researchers with an existing programming background should be be reasonably familiar with the concepts we use.

We have already successfully created an *LLVM* front-end adapter which wraps the ever-changing *LLVM API*, defined an interface for combining abstract value domains, and implemented two different versions of the value domain module. The design of Angie is now fairly stable and reflects many hours of discussions regarding the overall design of the framework.

2. Symbolic Memory Graphs

The following chapter introduces basics of *SMG*s described in [2].

2.1 Nodes and edges description

The idea behind *Symbolic Memory Graphs* is to express the memory configuration of a program as an oriented graph of individual memory areas with a certain degree of abstraction. This representation is used during *abstract interpretation*.

*SMG*s consists of two kinds of nodes: *objects* and *values*, where objects are further divided into *regions* and *list segments*.

Regions simply represents the individual memory areas while list segments are more abstract: each represent a part – uninterrupted segment – of single- or double-linked list.

There are also two types of edges: *has-value* and *points-to* edges.

2.2 Basics about operations

The first kind of operation is reinterpretation: data read, write, and join reinterpretation. On the one hand, because *SMG*s allow fields of single object to overlap (even to the degree they are different only in type) it is possible to create multiple views of the same memory area - hence the read reinterpretation. On the other hand, when writing to a field which overlaps with other fields, we need to reflect these changes to memory – that is what write reinterpretation is for.

*SMG*s are created "on-the-fly" during the analysis and the abstraction operation is triggered on certain conditions. Whenever this occurs, analysis tries to identify group of regions as - for example - single-linked list and converts appropriate regions to list-segment(s).

Abstraction and reinterpretation operations over *SMG*s are specified using a custom join operator. Join operator is the core of the entire abstraction: it is a binary operator, taking two *SMG*s as an input, outputting their possible join. Its primary usage is to lower the number of *SMG*s for every possible program state by merging both input *SMG* into (probably more general) one with restrictions on information loss in place. Details of its formal definition are beyond the scope of this work.

2.3 Example

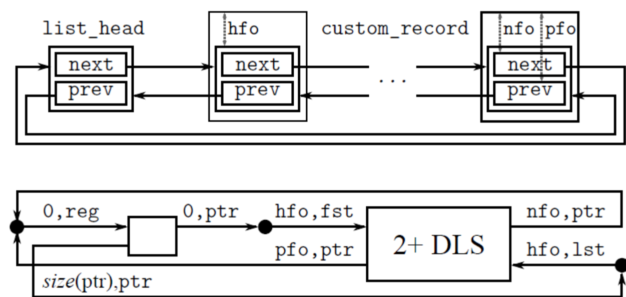

Figure 1. Standard memory view [top] and Abstract–SMG view [bottom]

Figure 1 shows and example of *SMG* corresponding to linux-like double linked list. The list consists of head and no-less than 2 elements - first and last. Head of the list is represented by a *region* in the left part of the graph and has two values of pointer type, at offests 0 and *size(ptr)*. Both corresponding points-to edges are pointing to the useful part of the list represented by 2+ DLS, one to the first and the second to the last element.

3. Important parts of Angie design

3.1 Main algorithm – the verification loop

All static program verifiers use some sort of program representation to perform their analysis. Angie uses *LLVM IR* (intermediate representation) as oppose to the predecessor tool Predator which was analysing program code in a form of GIMPL instructions. The *LLVM* has been chosen because of its strenghts— it allows for analysis of programs written in all clang compatible languages, it is easy to modify, filter, optimize and has strong support from development community.

The verification loop starts by loading *LLVM IR* code and initializing the analysis domain. The internal IR code can be modified/filtered to simplify the code analysis.

Angie uses the IR code to build simplified control-flow graph (CFG) whose nodes are our custom operations with arguments representing semantics of the analysed program.

The result of the initialization part of the verification loop is the root node of the CFG (entry point) and a zero state (initialized constants etc.)

The program analysis is a form of abstract interpretation, Angie walks the control-flow graph and performs the abstract operations resulting in a new state (in case of SMGs: memory representation of the current state of the program—SMG and Values). When the analyser encounters CFG node it has already visited, to prevent program state explosion a join operation over the old and a new state might be executed. Walking the control-flow graph is achieved by a container (either queue or list) containing pairs of CFG node and a last program state. At the begining of the analysis the root CFG node with the zero state is enqueued into the Worklist then in every step of the analysis an operation is executed from the top of the Worklist and a following node(s) with the new state is placed back to the Worklist. This is repeated in a loop until the Worklist is empty (from here the name verificatoin loop).

The verification loop is expected to be more generalized in the future to support other techniques beside abstract interpretation.

3.2 Analysis domain primitive

Angie models the analysis domain as an object representing a abstract program state which can be cloned and a set of free operations able to modify that state. This approach provides valuable flexibility towards possibility of creating new custom operations for diagnostic needs, forcing a specific action etc. without the necessity of radically alternating the main verification loop. This concept also allows for relative simple exchange/extension of the program analysis. See description of Figure 2.

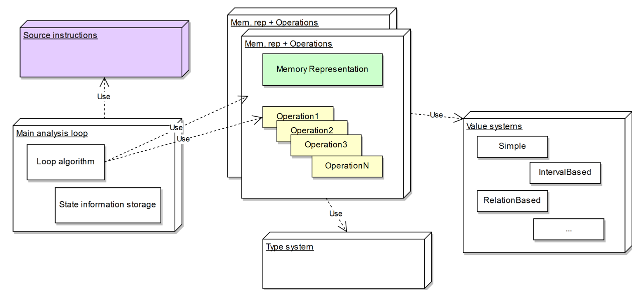

Figure 2. General concept of abstract analysis domain

3.3 Abstract value representation domain

An abstract value domain is a set of all possible values an abstract value can represent. The domain allows to share information with other domains effectively encapsulating internal representation of the abstract values and operations with them.

Almost every program analysis requires an abstract domain of some kind so the Angie framework already contains an abstract value domain implementation which provides simpler analysis development, however the framework does not tie developers hand behind his back and allows for the abstract value domain to be switched for another one.

The abstract domain implemented in the Angie framework supports constant integer and partly interval abstraction with arithmetic operations $+$, $-$, $*$, $/$, %, bit operations *shift, or, xor, and, not,* comparisons $==$, $!=$,$<$,$>$ between two abstract values, creating constants and unknown values, bit truncation and extensions. Logical operations and, not, or are not implemented since the *LLVM IR* code does not use them anyway. See Figure 3 for the current abstract domain interface.

4. The implementation

Angie is currently hosted on GitHub with continuous integration hooks and we provide it under the GNU LGPL licence v3+. The repository contains dependency and install description, has a stable master branch and unstable feature and develop branches.

All currently used third party libraries are header-only and licence compatible. One of those, Range-v3[12], is worth mentioning - it is based on the current version of Ranges and Concepts drafts for Technical Specifications[13]. It allows non-owning selection, trans-

[12] `https://github.com/ericniebler/range-v3`

[13] TS are papers for features that are proposed to be merged in one of the next ISO C++ standards.

<<*Interface*>>
IValueContainer

+ CreateVal (ValueId, Type): ValueId
+ CreateConstIntVal (uint64_t value, Type): ValueId
+ CreateConstFloatVal (double value, Type): ValueId

+ IsCmp (firstID, secondID, Type, CmpFlags): tribool
+ Is{True|False} (ValueId, Type): tribool
+ IsUnknown(ValueId): bool
+ GetAbstractionStatus(ValueId): AbstractionStatus
+ IsZero (ValueId): tribool

+ Cmp (firstID, secondID, Type, CmpFlags): ValueId
+ Assume (firstID, secondID, Type, CmpFlags): void
+ Assume{True|False} (ValueId): void

+ BinOp (firstID, secondID, Type, BinaryOpOptions): ValueId
+ BitNot(ValueId, Type): ValueId

+ ExtendInt (ValueId, sourceType, targetType, ArithFlags): ValueId
+ TruncateInt (ValueId, sourceType, targetType): ValueId

Figure 3. Abstract value domain interface

formation and also generator views to be constructed as single objects using functional-like syntax, enabling easy-to read constructs and re-use of algorithms similar to <functional>[14]. The project is targeting C++ 14 and is tested with a minimum of GCC 4.9 / MSVC 2015.

Currently implemented features are:

- Direct support for most LLVM IR critical constructs, rest can be simplified via pre-run transformation passes
- Symbolic Memory Graphs without abstraction
- Basic intrinsic functions for analysis and debugging

In progress:

- Basic doubly linked list abstraction for SMGs
- Simple analysis implementation example and tutorial

5. Conclusions

Nowadays the world has a high demand for a software whose correctness and proper operation is a question of life and death. This being a fact, a program verification methods have become a very important area of research.

To simplify the conditions for developers of new program analysers, the VeriFIT group is working on a framework, that has the potential to be the bread and water for students and enthusiastic researchers, allowing them to skip most of the waste code concerning loading the program into its intermediate representation, boilerplate code around the main verification loop and abstract value domain model and fully focus on what is really important – the science of creating a new analyser.

We are going to continue working on the implementation of *Symbolic Memory Graphs* analysis, refine the code of framework itself to more closely match the proposed design and provide proper documentation and examples.

Moreover, in the future, we hope to provide an interface to implement analysis in languages different than C++, get more people to maintain the project and even more people to create analysis / verification tools with it.

[14] A part of C++ standard library containing standard algorithm functions working with iterators

Acknowledgements

We would like to thank our fellow VeriFIT members: namely the project supervisor Tomáš Vojnar and the co-supervisor Petr Peringer for their help and cooperation on this project, also Petr Muller and Viktor Malík for their work on the previous halted attempt on the new *SMG* C++ implementation, and last but not least Veronika Šoková for her initial research on *LLVM IR* and its simplification transformations [6] .

References

[1] Veronika Šoková. Vývoj LLVM adaptéru pro infrastrukturu Code Listener. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2013. Details: http://www.fit.vutbr.cz/study/DP/BP.php?id=15896.

[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] P. Muller, P. Peringer, and T. Vojnar. *Predator Hunting Party (Competition Contribution)*. In *Proc. of TACAS'15 as a competition contribution within SV-COMP'15*, *LNCS* 9035, pages 443–446, Springer, 2015.

[4] M. Kotoun, P. Peringer, Veronika Šoková, and T. Vojnar. *Optimized PredatorHP and the SV-COMP Heap and Memory Safety Benchmark (Competition Contribution)*. In *Proc. of TACAS'16 as a competition contribution within SV-COMP'16*, *LNCS* 9636, pages 942–945, Springer, 2016. An extended version is available here: http://www.fit.vutbr.cz/~vojnar/Publications/predators-svcomp-16.pdf.

[5] D. Beyer. *Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016).* Proc. TACAS, Springer, 2016.

[6] Veronika Šoková. Analýza práce s dynamickými datovými strukturami v c programech. Master's thesis, Vysoké učení technické v Brně, Fakulta informačních technologií, 2015. Details: http://www.fit.vutbr.cz/study/DP/DP.php?id=18790.