VeriAbsL: Scalable Verification by Abstraction and Strategy Prediction (Competition Contribution)

Priyanka Darke[1,⋆(✉)], Bharti Chimdyalwar[1], Sakshi Agrawal[1], Shrawan Kumar[1], R Venkatesh[1], and Supratik Chakraborty[2]

[1] TCS Research, Pune, India
priyanka.darke@tcs.com
[2] Indian Institute of Technology, Bombay, India

Abstract. We present VeriAbsL, a reachability verifier that performs verification in three stages. First, it slices the input code using a combination of two slicers, then it verifies the slices using *predicted* strategies, and at last, it composes the result of verifying the individual slices. We introduce a novel *shallow slicing* technique that uses variable reference information of the program, and data and control dependencies of the entry function to generate slices. We also introduce a novel *strategy prediction* technique that uses machine learning to predict a strategy. It uses boolean features to describe a program to a neural network that predicts a strategy. We use the portfolio of VeriAbs, a reachabiltiy verifier with manually defined strategies. In SV-COMP 2023, VeriAbsL verified 227[3] more programs than VeriAbs, and 475[3] programs that VeriAbs could not verify.

1 Verification Approach

It is folklore in automated software verification that no single verification technique is good enough to verify all programs of interest. This limitation led to the advent of strategy selection-based verifiers that use predefined verification strategies [4]. A strategy is a sequence of verification techniques applied to a program, where each technique is bounded by a heuristically defined time limit. In this paper, we present a strategy prediction-based reachability verifier for C programs called VeriAbsL. It verifies a program in stages using a portfolio of two slicing, and ten verification techniques. First, it slices a program using a sequence of slicers. Then it uses a few syntactic and semantic features of the slice to predict a strategy and verify the slice. Lastly, it composes the result of verifying each slice. VeriAbsL uses a sequential combination of two slicers, a *slicer-analyzer* [7], and a novel *shallow slicer* or SSLICER. SSLICER is applied to programs that could not be sliced by the slicer-analyzer. The slicer-analyzer is more efficient than SSLICER, but applies to a smaller class of programs as explained in Section 1.2. Let a program P be sliced into n slices. A strategy prediction module extracts the features of each slice P_i, $1 \leq i \leq n$, and predicts a strategy for it using a neural network. The program P is safe if each slice P_i is safe, and P is unsafe if any slice P_i is unsafe. If program P cannot be sliced, then a

⋆ P. Darke—Jury member
[3] Without witness validation.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 588–593, 2023.
https://doi.org/10.1007/978-3-031-30820-8_41

strategy is predicted for P itself. Fig. 1 shows the architecture of VeriAbsL. As shown VeriAbsL uses the portfolio of a *strategy selection*-based verifier called VeriAbs [7].

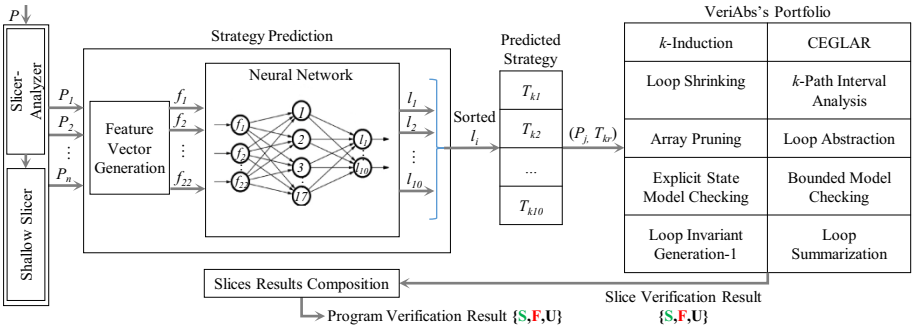

Fig. 1. VeriAbsL Architecture (**S**: Program Safe, **F**: Property Fails, **U**: Unknown)

1.1 Strategy Prediction using Machine Learning (ML)

Despite the advantages of sequencing multiple verification techniques in a strategy, experimental evidence indicates that each strategy works well for only a class of programs. When a new class is encountered, experts define a new strategy and update the strategy-selection algorithm of the verifier. This is a tedious task. In order to automate it, recently ML-based verifiers have been used with partial success [5]. VeriAbsL is one such verifier. It uses a simple ML-based approach explained as follows.

Feature Vector Generation. VeriAbsL uses a feature vector $\boldsymbol{f}$ of 22 boolean features that describe a few semantic, or syntactic constructs of the input slice P_j. For example, a boolean feature $f_i \in \boldsymbol{f}$ if set to *true* can indicate the presence of arrays in the input code, and *false* can indicate that no arrays are used. These features are computed using a light-weight static analysis, and derived from those presented in [8].

Neural Network. VeriAbsL uses a three layered neural network with multi-class classification, one class for each of the ten techniques in our portfolio. It has 22, 17, and 10 neurons in the respective layers. It was trained using ReLU for the hidden layer and softmax for the output layer, as activation functions, and with the mean-squared error loss function. It translates an input feature vector $\boldsymbol{f}$ representing program slice P_j into likelihoods of success l_i, $1 \leq i \leq 10$, of the corresponding verification techniques T_i in the portfolio for slice P_j. Each output node of the neural network n_i represents one verification technique T_i and the value l_i generated by the network at that node n_i is a heuristic measure of the relative likelihood that technique T_i will successfully verify/disprove the property for slice P_j within 900 seconds.

Strategy Prediction. A strategy $(T_{k_1},...,T_{k_{10}})$, $1 \leq k_r \leq 10$, $1 \leq r \leq 10$, is created by sorting the relative likelihoods of success l_i of each verification technique T_i in the decreasing order. The techniques T_i are invoked in that order to verify slice P_j.

Experimental Results. The neural network in VeriAbsL was trained on 800 randomly selected SV-COMP 2022 ReachSafety benchmarks. At SV-COMP 2023 out of all 6138 benchmarks, VeriAbsL verified 227 more programs in 4.4% lesser time than

VeriAbs[4] and verified 475 programs that VeriAbs could not verify[3]. This was because VeriAbsL predicted useful techniques early in its strategies, while VeriAbs selected unsuitable strategies and ran out of time. Further the randomly selected training data did not contain any benchmarks from three ReachSafety sub-categories namely Combinations, ProductLines, and Hardware. VeriAbsL verified 72 more programs than VeriAbs in these 3 sub-categories demonstrating that strategy-prediction in VeriAbsL generalizes to programs for which it was not trained. VeriAbsL ran out of time for 248 programs verified by VeriAbs because the randomly selected training data did not contain any sample corresponding to two techniques, namely VAJRA [6] and Counter-Example Guided Loop Abstraction Refinement (CEGLAR) [4], needed to verify the 248 programs. Thus they were always predicted late. Further VeriAbsL verified 1047 and 543 more benchmarks compared to the other ML-based strategy prediction tools, GRAVES [11] and PESCO [12], respectively.

Strengths and Weaknesses of Strategy Prediction. VeriAbsL can verify more programs than VeriAbs in spite of the same portfolio because it uses ML for strategy prediction. Also VeriAbsL demonstrates that a small set of boolean features can be used successfully to verify programs, while other successful verifiers predict a strategy using graph based learning methods [12]. Further VeriAbsL does not incorporate a feedback mechanism that can penalize a technique if it cannot verify a program. Such a feedback mechanism can improve its efficiency and accuracy.

1.2 Shallow Slicer

SSLICER is a generalization of the slicer-analyzer presented in [7] and like the latter, aims for a scalable slicing with respect to calls in entry function *main*. But unlike the slicer-analyzer, SSLICER allows multiple calls in *main* to (1) refer to the same global variable, (2) transitively invoke the same function, or (3) have transitive dependence on the same data element or control structure in *main*.

SSLICER partitions the program functions directly or indirectly called from *main* into n sets $F_1 ... F_n$ such that the following conditions, termed as *partition-independence*, are satisfied: (1) Each partition F_i contains at least one function directly called from *main*. (2) Each partition F_i contains functions which are either directly or transitively called from *main*. (3) All functions transitively called from function $f \in F_i$ also belong to F_i, the same partition as f. Thus if $T(f)$ is the set of functions transitively called from f, then $\forall i,\ 1 \leq i \leq n,\ \forall f \in F_i,\ T(f) \subseteq F_i$. (4) No two functions $f \in F_i$ and $g \in F_j$ belonging to different partitions transitively call the same function or refer to the same global variable. Let $V(F_i)$ be the set of global variables referred to by functions in set F_i then $\forall i,j \mid 1 \leq i \leq n,\ 1 \leq j \leq n,\ i \neq j \implies (V(F_i) \cap V(F_j) = \emptyset)$ (5) Let $main_i$ be the function generated when a program containing only one function, the function *main*, is sliced (using known slicing techniques [9]) with respect to calls to functions in set F_i which are directly called from *main*. Then functions of no other set F_j, $i \neq j$, should refer to the variables used in $main_i$. Thus $\forall i,j \mid 1 \leq i \leq n,\ 1 \leq j \leq n,\ i \neq j \implies (V(main_i) \cap V(F_j) = \emptyset)$ (6) n is the largest possible natural number satisfying the above conditions.

[4] The competition score of VeriAbs is greater than VeriAbsL because of 8 incorrect results produced due to bugs in the implementation of a technique predicted by VeriAbsL. This technique was not executed for these 8 programs by VeriAbs.

A slice P_i corresponding to each set F_i is generated. The set of functions in slice P_i is given by $main_i \cup F_i$. To create the slice, call graph and referred variables information is computed using call-trees, and a light-weight flow-insensitive pointer analysis. We assume that function *main* itself is not a part of any recursive call chain, and does not specify the assertions directly.

```
main(){
    if(a&&b)f1();
    f2();
    if(b) f5();
}
f1() { f3(); }
f2() { f4(); }
f3() { c++; }
f4() { a++; }
f5() { d++; }
```

(a) Input Code

```
main(){
    if(a&&b)f1();
    f2();
}
f1() { f3(); }
f2() { f4(); }
f3() { c++; }
f4() { a++; }
```

(b) Slice 1

```
main(){
    if(b) f5();
}
f5() { d++; }
```

(c) Slice 2

Fig. 2. Example

Example. Consider the program presented in Fig. 2a. In this example functions called from *main* can be initially partitioned into three sets {`f1, f3`}, {`f2, f4`,} and {`f5`} as `f1` calls `f3`, `f2` calls `f4`, and `f5` does not refer to any function or variable that other functions refer to. But function `f4` refers to variable `a`. If a program containing only the body of function `main` shown in Fig. 2a were to be sliced with respect to the call to `f1` in `main` then it would refer to variable `a`. Function `f1` belongs to the first partition and `f4` to the second. To satisfy the fifth condition of *partition-independence* functions `f1` and `f4` must belong to a single partition. Thus finally there are two partitions - {`f1, f2, f3, f4`}, and {`f5`}. The slices created for the first and second partitions are shown in Figures 2b and 2c respectively. Notice that since function `f5` does not refer to variable `b` in its body, it need not be merged with the other partition even though the body of sliced `main` in Fig. 2c refers to variable `b`.

Experimental Results. We compare the performances of VeriAbsL with (1) *slicer-analyzer*, and (2) *slicer-analyzer* and SSLICER, on all 6138 benchmarks of the Reach-Safety category of SV-COMP 2023. The first configuration generated slices for 671 programs while the second generated slices for 1369 programs showing better applicability. Further, due to SSLICER, VeriAbsL terminated its analysis for 42 more programs, showing improved scalability, and its portfolio could verify 4 additional programs.

1.3 Software Project, Architecture, and Setup

The Foundations of Computing research group at TCS Research [1] has developed VeriAbsL. It is written in Perl, Java and Python. It uses TCS's program analysis framework [10] for static analysis, and TensorFlow libraries [2] for learning. VeriAbsL uses VeriAbs's portfolio [7], except VAJRA [6] because it is not supported on Ubuntu 22.04 LTS. VeriAbsL participated in the Reach-Safety category at SV-COMP 2023, and is available at [3]. The installation instructions are in `VeriAbsL/INSTALL.txt`, the BenchExec[5] wrapper script for the tool is `veriabsl.py`, and the benchmark definition file is `veriabsl.xml`. On successful verification, VeriAbsL generates a witness in the current working directory as `witness.graphml`. A sample command to verify property given in file `reach-safety.prp` for a program, given in `a.c`, of a 32-bit (or 64-bit) architecture is as follows: `VeriAbsL/scripts/veriabs -32|64 --property-file reach-safety.prp a.c`

[5] https://github.com/sosy-lab/benchexec

2 Data-Availability Statement

VeriAbsL is available as part of SV-COMP 2023 verifier repository at https://gitlab.com/sosy-lab/sv-comp/archives-2023/-/blob/main/2023/veriabsl.zip. For any queries please contact the authors at veriabs.tool@tcs.com.

References

1. Foundations of Computing Group at TCS Research. https://www.tcs.com/what-we-do/research.
2. TensorFlow. https://www.tensorflow.org/.
3. VeriAbsL Tool Archive. https://gitlab.com/sosy-lab/sv-comp/archives-2023/-/blob/main/2023/veriabsl.zip.
4. M. Afzal, A. Asia, A. Chauhan, B. Chimdyalwar, P. Darke, A. Datar, S. Kumar, and R Venkatesh. VeriAbs: Verification by Abstraction and Test Generation. In *ASE*, pages 1138–1141, 2019.
5. Dirk Beyer. Progress on software verification: SV-COMP 2022. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 375–402. Springer, 2022.
6. Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs with full-program induction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 22–39. Springer, 2020.
7. P. Darke, S. Agrawal, and R. Venkatesh. VeriAbs: A Tool for Scalable Verification by Abstraction (Competition Contribution). In *Proc. TACAS (2)*, LNCS 12652. Springer, 2021.
8. Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. Empirical software metrics for benchmarking of verification tools. In Jens Knoop and Uwe Zdun, editors, *Software Engineering 2016*, pages 67–68, Bonn, 2016. Gesellschaft für Informatik e.V.
9. Mark Harman and Robert M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.
10. S. Khare, S. Saraswat, and S. Kumar. Static program analysis of large embedded code base: an experience. In *ISEC*, pages 99–102, 2011.
11. Will Leeson and Matthew B. Dwyer. Graves-cpa: A graph-attention verifier selector (competition contribution). In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 440–445, Cham, 2022. Springer International Publishing.
12. Cedric Richter, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Algorithm selection for software validation based on graph kernels, 2020. https://link.springer.com/article/10.1007/s10515-020-00270-x.
13. Cedric Richter and Heike Wehrheim. Pesco: Predicting sequential combinations of verifiers. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–233, Cham, 2019. Springer International Publishing.