Predicting Rankings of Software Verification Tools[*]

Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, Heike Wehrheim
Department of Computer Science
Paderborn University,
Germany

ABSTRACT

Today, software verification tools have reached the maturity to be used for large scale programs. Different tools perform differently well on varying code. A software developer is hence faced with the problem of *choosing* a tool appropriate for her program at hand. A *ranking* of tools on programs could facilitate the choice. Such rankings can, however, so far only be obtained by running all considered tools on the program.

In this paper, we present a machine learning approach to *predicting* rankings of tools on programs. The method builds upon so-called label ranking algorithms, which we complement with appropriate *kernels* providing a similarity measure for programs. Our kernels employ a graph representation for software source code that mixes elements of control flow and program dependence graphs with abstract syntax trees. Using data sets from the software verification competition SV-COMP, we demonstrate our rank prediction technique to generalize well and achieve a rather high predictive accuracy (rank correlation > 0.6).

CCS CONCEPTS

•**Computing methodologies** →**Ranking;** *Support vector machines;* Cross-validation; •**Software and its engineering** →Software verification; Formal software verification;

KEYWORDS

Software verification, machine learning, ranking.

ACM Reference format:
Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, Heike Wehrheim . 2017. Predicting Rankings of Software Verification Tools. In *Proceedings of 3rd International Workshop on Software Analytics, Paderborn, Germany, September 2017 (SWAN'17),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Over the past years, software analytics [8] has become a very active research area. Software analytics is concerned with analysing software source code and its associated artefacts like code reviews, documentation and bug reports. Such data can today be found in numerous repositories, and analysis can provide insights into various sorts of questions in software development, ranging from program understanding over defect prediction to power consumption.

In this paper, we employ software analytics to get an answer to the question "What software verification tool do I use for showing correctness of my program?". Software verification tools today are numerous, and the past years have seen the tuning of tools to performance and precision, but still not every tool is suitable for every program and property at hand. Software verification competitions provide a comparison of tools on the basis of given benchmark verification tasks, i.e., programs together with properties. In the area of automatic verification, the most prominent competition today is the annual Competition on Software Verification SV-COMP [1]. The outcome of SV-COMP are *rankings of tools* (overall and per categories) computed by means of a scoring schema. These annually published ranking results provide data about the relative performance of tools and is readily usable for software analytics. Based on this data, a *prediction* of a likely ranking of tools on a given new verification task becomes possible. In this paper, we develop a method for performing such rank predictions.

Our method for rank prediction builds upon machine learning methods for the so-called *label ranking* problem; more specifically, we make use of *ranking by pairwise comparison* [6] using support vector machines as base learners. The key ingredient of our approach is the definition of suitable *kernel functions* [10], which act as similarity measures on verification tasks. So far, two other machine learning methods for selecting (albeit not ranking) tools or algorithms for verification have been proposed [4, 12], both of them being based on feature vectors explicitly capturing structural features of programs (like number of variables, loops, etc.). With our kernels, we take a different approach: we supply the learning algorithm with a representation of source code that enables the learner itself to identify the distinguishing patterns. We believe that our kernels are thus more readily usable for other program analysis tasks.

We have implemented our technique and carried out experimental (cross-validation) studies using data from SV-COMP 2015 and 2017. The experiments show that our technique can predict rankings with high accuracy.

2 REPRESENTING VERIFICATION TASKS

Our objective is to *predict* rankings of software verification competitions via machine learning. To this end, the learning algorithm has to be supplied with training data. We start with explaining what kind of data our rank prediction technique is supplied with, namely verification tasks and rankings.

Definition 2.1. A *verification task* (P, φ) consists of a program P (for SV-COMP written in C) and a property (also called specification) φ (typically written as assertion into the program).

[*] This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

SWAN'17, Paderborn, Germany

DOI: 10.1145/nnnnnnn.nnnnnnn

```
1  int i;          6  i = 0;
2  int n;          7  while (i <= n)
3  int sn;         8        sn = sn + 2;
4  n = input();    9        i = i + 1;
5  sn = 0;         10 assert (sn == n*2 || sn == 0);
```

Figure 1: The verification task P_{SUM}

Figure 1 shows our running example P_{SUM} of a verification task (computing n times 2 via addition). In a verification run, a verification tool is run on a verification task in order to determine whether the program fulfills the specification. The outcome of such a verification run is a pair (TIME, ANSWER), where TIME is the time in seconds from the start of the verification run to its end, and ANSWER is either TRUE, FALSE or UNKNOWN. The ranking is done via a scoring schema which gives positive and negative points to outcomes. When the scores of two tools are the same, the runtimes (of successful runs) determine the ordering.

The purpose of the machine learning algorithm is to learn from the provided ranking how tools will perform on specific verification tasks. Our machine learning technique is based on *kernel methods* (see e.g. [10]). In general, a kernel can be interpreted as a similarity measure on data instances (in our case verification tasks), with the idea that similar results (in our case rankings) are produced for similar instances. While kernel-based learning algorithms are completely generic, the kernel function itself is application-specific.

The simplest way of defining a kernel is via the inner product of *feature vectors*, i.e., vectorial representations of data objects. In the two approaches existing so far [4, 12], corresponding features of programs, such as number of loops, are defined in an explicit way. Our approach essentially differs in that features are specified in a more indirect way, namely by systematically extracting (a typically large number of) generic features from a suitable representation of the verification task. Selecting the useful features and combining them appropriately is then basically left to the learner. For example, if data objects are graphs, a feature may correspond to a subgraph, and the value of the feature is 1 if the subgraph is present and 0 otherwise. In our case, generic features are subgraphs of a certain depth in our graph representation of the program. As an example, program P_{SUM} (amongst others) contains a loop with two assignments.

But how to represent the verification tasks? The first idea is to use the source code itself (i.e., strings). However, the source code of two programs might look very different although the underlying program is actually the same (different variable names, `while` instead of `for` loops, etc.). What we need is a representation that abstracts from such issues but still represents the structure of programs, in particular dependencies between elements of the program. These considerations (and some experiments comparing different representations) have led to a *graph* representation of programs combining concepts of three existing program representations:

Control flow graphs: CFGs record the control flow in programs and thus the overall structure with loops, conditionals etc.; these are needed, for example, to see loops.

Program dependence graphs: PDGs [5] represent control and data dependencies between elements in programs. This information is important, e.g., to detect a loop boundary depending on an input variable (as in program P_{SUM}).

Abstract syntax trees: ASTs reflect the syntactical structure of programs according to a given grammar and can for instance help to reveal the complexity of expressions.

Unlike CFGs and PDGs but (partly) alike ASTs, we abstract from concrete names occurring in programs. Nodes in the graph will thus not be labelled with statements or variables as occurring in the program, but with abstract identifiers. We let *Lab* be the set of all such labels.

Definition 2.2. Let P be a verification task. The *graph representation* of P is a graph $G = (N, E, s, t, \rho, \tau, \eta)$ with

- N a set of nodes (basically, we build an AST for every statement in P, and use the nodes of these ASTs),
- E a set of edges, with $s : E \to N$ denoting the start and $t : E \to N$ the end node of an edge,
- $\rho : N \to Lab$ a labelling function for nodes,
- $\tau : E \to \{CD, DD, SD, CF\}$ a labelling function for edges reflecting the type of dependence: *CD* (control dependency) and *DD* (data dependency) origin in PDGs, *SD* (syntactical dependence) is the "consists-of" relationship of ASTs and *CF* (control flow) the usual control flow in programs,
- $\nu : E \to \{T, F\}$ a function labelling control dependence edges according to the valuation of the conditional they arise from. All other edges are labelled true.

We let $\mathcal{G}_V$ denote the set of all verification task graphs.

Figure 2 depicts the graph representation of the verification task P_{SUM}. The rectangle nodes represent the statements in the program and act as root nodes of small ASTs. For instance, the rectangle labelled `Assert` at the bottom, middle represents the assertion in line 10. The gray ovals represent the AST parts below the root nodes. We define the *depth* of nodes n, $d(n)$, as the distance of a node to its root node. As an example, the depth of the `Assert`-node itself is 0, the depth of both ==-nodes is 2.

3 PREDICTING RANKINGS

This section starts with a short description of label ranking and the method of ranking by pairwise comparison. Moreover, we introduce our kernel functions on verification tasks.

3.1 Label Ranking

In the setting of label ranking (see e.g. [6]), we are interested in predicting rankings (total orders) $\succ$ on a finite set of K alternatives identified by *class labels* $\mathcal{Y} = \{y_1, \ldots, y_K\}$; in our case, the alternatives correspond to the verification tools to be compared. Formally, a total order $\succ$ can be identified with a permutation π of the set $[K] = \{1, \ldots, K\}$. Preferences on $\mathcal{Y}$ are "contextualized" by instances $x \in \mathcal{X}$, where $\mathcal{X}$ is an underlying instance space; in our case, instances are programs to be verified. Thus, each instance x is associated with a ranking $\succ_x$ of the label set $\mathcal{Y}$.

The goal in label ranking is to learn a "label ranker", that is, a model $\mathcal{M} : \mathcal{X} \to \mathbb{S}_K$, where $\mathbb{S}_K$ denotes the class of permutations of $[K]$. The predictions $\hat{\pi} = \mathcal{M}(x)$ produced by such a model for an instance x are evaluated in terms of an accuracy measure such

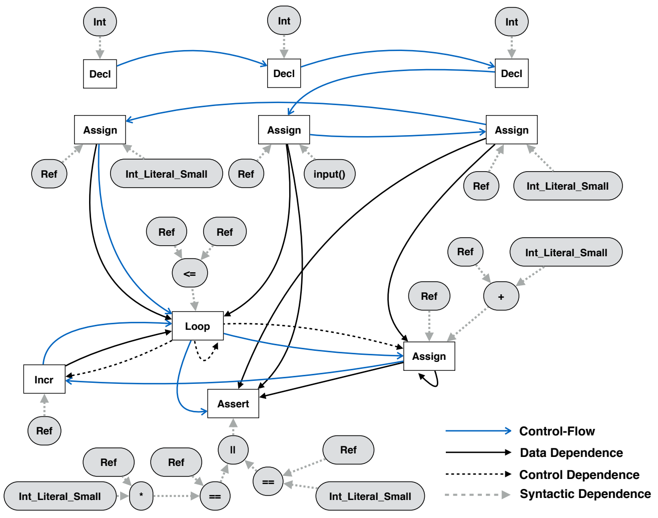

Figure 2: Graph representation of P_{SUM} eliding labelling ν, `Int_Literal_Small` representing an integer literal in [0,10]

as the *Spearman rank correlation*:

$$S(\pi,\hat{\pi}) = 1 - \frac{6\ \sum_{i=1}^{K}(\pi(i)-\hat{\pi}(i))^2}{K(K^2-1)} \in [-1,1]$$

As training data $\mathbb{D}$, a label ranker uses a set $\{(x_n,\pi_n)\}_{n=1}^{N}$ of instances with associated rankings.

3.2 Ranking by Pairwise Comparison

We tackle the label ranking problem using the method of *ranking by pairwise comparison* (RPC), a meta-learning technique that reduces a label ranking task to a set of binary classification problems [6]. More specifically, the idea is to train a separate model $\mathcal{M}_{i,j}$ for each pair of labels $(y_i,y_j) \in \mathcal{Y}$, $1 \le i < j \le K$, and to combine the predictions of these models into an overall ranking.

The $\mathcal{M}_{i,j}$ are binary classifiers, which, given an instance x, are supposed to predict 1 if $y_i \succ_x y_j$ and 0 if $y_j \succ_x y_i$. In our approach, we train these classifiers using support vector machines [10], extracting the binary target values from the rankings observed for the training instances. The final ranking is obtained by sorting the labels y_i in decreasing order of their Borda-scores $\sum_{j \neq i} M_{i,j}(x)$.

SVMs are so-called "large margin" classifiers separating positive from negative training instances in $\mathbb{R}^m$ by means of a linear hyperplane. They can be turned into flexible nonlinear classifiers using the idea of "kernelization", which requires a kernel function on $\mathcal{X}$.

Definition 3.1. A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *positive semi-definite kernel* iff k is symmetric, i.e., $k(x,x') = k(x',x)$, and

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j k(x_i,x_j) \ge 0$$

for arbitrary N, arbitrary instances $x_1,\ldots,x_N \in \mathcal{X}$ and arbitrary $c_1,\ldots,c_N \in \mathbb{R}$.

Next, we address the question of how to define appropriate kernel functions on verification tasks.

Algorithm 1 relabel (Graph relabelling)

```
Input: $G = (N, E, s, t, \rho, \tau, \nu)$ graph
    $z : \Sigma^* \to \Sigma$ injective compression function
    $\eta : N \to 2^E$ neighbour function
    $m$ iteration bound
Output: relabelled graph $G$
for $i = 1$ to $m$ do
    for $n \in N$ do
        $Aug(n) := \big\langle z\big(\rho(s(e)) \oplus \tau(e) \oplus \nu(e)\big) \mid e \in \eta(n)\big\rangle$
        $Aug(n) := sort(Aug(n))$
        $str(n) := concat(Aug(n))$
        $str(n) := \rho(n) \oplus str(n)$
        $\rho(n) := z(str(n))$
return $G$
```

3.3 Graph Kernels for Verification Tasks

Verification tasks are represented by specific graphs, whence our kernel needs to operate on graphs. As our graphs are representations of programs with several thousand lines of code, we have chosen to proceed from our own kernel development based on *Weisfeiler-Lehman subtree kernels* [11], which are known to scale well to large graphs.

Weisfeiler-Lehman (WL) kernels are extensions of the WL test of isomorphism between two discretely labelled, undirected graphs [13]. This test basically compares graphs according to their node labels. For taking edges into account, node labels are extended with information about neighbouring nodes in three steps: Concatenate label of node n with labels of its neighbours (Augmentation), sort this sequence according to predefined order on labels (Sorting), compress thus obtained sequences into new labels (Compression). These steps are repeated until the node label sets of the two graphs differ or a predefined bound on the number of iterations is exhausted.

For making this WL test act as a kernel for verification tasks, we made three adaptations to the graph relabelling, giving rise to Algorithm 1: (1) extension to directed multigraphs, (2) customization to specific neighbours of nodes, and (3) integration of edge labels. In Algorithm 1, we use the notation $\langle \ldots \mid \ldots \rangle$ for list comprehensions, defining a sequence of values. We let $\Sigma = \mathbb{N}$ with the usual ordering $\le$ and map all node identifiers and edge labels to $\mathbb{N}$. The functions *sort* and *concat* sort sequences of labels (in ascending order) and concatenate sequences, respectively.

Definition 3.2. Let $G_i = (N_i, E_i, s_i, t_i, \rho_i, \tau_i, \nu_i)$, $i = 1,2$ be graph representations of verification tasks, $z : \Sigma^* \to \Sigma$ a compression function, $m \in \mathbb{N}$ an iteration bound, $d \in \mathbb{N}$ a depth for subtrees and $\eta_i : N_i \to 2^{E_i}$ neighbour functions. The *verification graph kernel* $k_{\eta_1,\eta_2,z}^{(d,m)} : \mathcal{G}_V \times \mathcal{G}_V \to \mathbb{R}$ is defined as

$$k_{\eta_1,\eta_2,z}^{(d,m)}(G_1,G_2) = \sum_{i=1}^{m} k^d \begin{pmatrix} relabel(G_1,z,\eta_1,m), \\ relabel(G_2,z,\eta_2,m) \end{pmatrix}$$

with

$$k^d(G,G') \quad = \quad \sum_{n\in N}\sum_{n'\in N'} k_\delta^d(n,n') \text{ and}$$

$$k_\delta^d(n,n') \quad = \quad \begin{cases} \delta(\rho_1(n),\rho_2(n')) & \text{if } d(n) \le d \wedge d(n') \le d \\ 0 & \text{else} \end{cases},$$

Table 1: SV-COMP 2017 – Prediction accuracy (mean and standard deviation) in terms of the Spearman rank correlation

Kernel / Data Set	SAFETY	TERMINATION	MEMSAFETY
k_{CF} (CFG)	**.635 ± .003**	.657 ± .007	.755 ± .004
k_{DD} (data dependency)	.618 ± .003	.635 ± .006	.754 ± .007
k_{CD} (control dependency)	.627 ± .002	.637 ± .006	.756 ± .005
$k_{CD,DD}$ (PDG)	.630± .005	.644 ± .003	**.757 ± .007**
$k_{CF,CD,DD}$ (PDG + CFG)	.632 ± .004	.658 ± .009	.756 ± .003
weighted combination	.634 ± .003	**.664 ± .010**	.756 ± .003
features of Demyanova et al. [4]	.560 ± .004	.560 ± .006	.717 ± .001
default predictor	.452 ± .003	.339 ± .004	.668 ± .001

where δ is a Dirac kernel defined as $\delta(u, w) = 1$ if u equals w and 0 otherwise.

THEOREM 3.3. *The function $k^{(d,m)}_{\eta_1,\eta_2,z}$ as defined above is a kernel, i.e., it is positive and semi-definite.*

Our kernels can now be used in a support vector machine within the ranking by pairwise composition approach outlined above.

4 EXPERIMENTAL EVALUATION

We have implemented the above described technique as to evaluate the performance of our method for rank prediction. Here, we report on SV-COMP 2017 data. We compared six variants of our kernel with respect to prediction accuracy, each of which focuses on different aspects of a program. Such kind of customization of kernels becomes possible thanks to the two neighbouring functions η_1 and η_2. In our case, neighbours are chosen according to the type of edge connecting them. We define $\eta_\ell, \ell \in \{CD, DD, SD, CF\}$ to be $\eta_\ell(n) = \{e \mid \tau(e) = \ell \wedge t(e) = n\}$, and let $\eta_L(n) = \bigcup_{\ell \in L} \eta_\ell(n)$ for a node n. For our kernels, we always use the same neighbouring function on both graphs. Hence, we will just use the edge labels employed in neighbouring functions as indizes for kernels. Our experiments include kernels $k_{\{CF\}}$, $k_{\{CD\}}$, $k_{\{DD\}}$, $k_{\{CD,DD\}}$, and $k_{\{CF,CD,DD\}}$. In addition, we included an equally weighted linear combination k_{lin} of some of our kernels, which is defined as

$$k_{lin}(G_1, G_2) = \tfrac{1}{3}k_{\{CF\}}(G_1, G_2) + \tfrac{1}{3}k_{\{CD\}}(G_1, G_2) + \tfrac{1}{3}k_{\{DD\}}(G_1, G_2)$$

To get an insight on how the prediction accuracy performs compared to state-of-the-art approaches, we also included the accuracy achieved by using the feature vectors from Demyanova et al. [4]. In addition, we constructed a *default predictor* for comparison: the default predictor takes all rankings of the data set used for learning, determines the ranking which minimizes the distance (wrt. Spearman rank correlation) to these rankings and always predicts this default ranking without any learning.

We constructed the following data sets for our experiments: SAFETY, TERMINATION, and MEMSAFETY. Each data set consists of several verification tasks (up to 500) and participating tools (up to 11) and is taken from the SV-COMP 2017 benchmark sets. To examine the prediction accuracy for each configuration, we performed a 10-fold cross-validation. The overall *accuracy* is then the average over all the accuracies encountered in each step.

In Table 1, we report the average prediction accuracies (and standard deviations) in terms of the Spearman rank correlation; note that an average accuracy of 0 would be obtained by guessing rankings at random, while +1 stands for predictions that perfectly coincide with the true ranking (and −1 for completely reversing that ranking). As can be seen, our approach shows a rather strong predictive performance. Depending on the verification task, different kernels achieve the best results, though the differences in performance are statistically non-significant. More importantly, our approach significantly outperforms the one of Demyanova et al. [4] as well as the default predictor on all tasks.

5 CONCLUSION

In this paper, we have proposed a method for predicting rankings of verification tools on given programs. Such rankings can be used by developers to choose a tool for a program or for building portfolio solvers. Our rank prediction technique builds on existing methods for label ranking via pairwise comparison. To this end, we have developed an expressive representation of source code, capturing various forms of dependencies between program elements.

Our approach can be seen as a tool for *algorithm selection*, a problem that has also been tackled by other authors [4, 12, 14]. The use of Weisfeiler-Lehman kernels has been studied by Sahs and Khan [9] and Li et al. [7]. Corazza et al. [3] employ kernels on programs for clone detection (however, tree kernels on ASTs, not graph kernels). A machine learning approach to software verification itself has recently been proposed by Chen et al. [2]. However, to the best of our knowledge, the use of machine learning for predicting *rankings* of verification tools has never been tried so far.

REFERENCES

[1] Dirk Beyer. 2015. Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015). In *TACAS (LNCS)*, Christel Baier and Cesare Tinelli (Eds.), Vol. 9035. Springer, 401–416.

[2] Yu-Fang Chen, Chiao Hsieh, Ondrej Lengál, Tsung-Ju Lii, Ming-Hsien Tsai, Bow-Yaw Wang, and Farn Wang. 2016. PAC learning-based verification and model synthesis. In *ICSE*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 714–724.

[3] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. 2010. A Tree Kernel based approach for clone detection. In *ICSM*. IEEE, 1–5.

[4] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. 2015. Empirical Software Metrics for Benchmarking of Verification Tools. In *CAV (LNCS)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 561–579.

[5] Susan Horwitz and Thomas W. Reps. 1992. The Use of Program Dependence Graphs in Software Engineering. In *ICSE*, Tony Montgomery, Lori A. Clarke, and Carlo Ghezzi (Eds.). ACM Press, 392–411.

[6] E. Hüllermeier, J. Fürnkranz, W. Cheng, and K. Brinker. 2008. Label Ranking by Learning Pairwise Preferences. *Artificial Intelligence* 172 (2008), 1897–1917.

[7] Wenchao Li, Hassen Saidi, Huascar Sanchez, Martin Schäf, and Pascal Schweitzer. 2016. Detecting Similar Programs via The Weisfeiler-Leman Graph Kernel. In *Software Reuse: Bridging with Social-Awareness (LNCS)*, Georgia M. Kapitsaki and Eduardo Santana de Almeida (Eds.), Vol. 9679. Springer, 315–330.

[8] Tim Menzies and Thomas Zimmermann. 2013. Software Analytics: So What? *IEEE Software* 30, 4 (2013), 31–37.

[9] Justin Sahs and Latifur Khan. 2012. A Machine Learning Approach to Android Malware Detection. In *EISIC*, Nasrullah Memon and Daniel Zeng (Eds.). IEEE Computer Society, 141–147.

[10] B. Schölkopf and AJ. Smola. 2001. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press.

[11] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research* 12 (2011), 2539–2561.

[12] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V. Nori. 2014. MUX: algorithm selection for software model checkers. In *MSR*, Premkumar T. Devanbu, Sung Kim, and Martin Pinzger (Eds.). ACM, 132–141.

[13] Boris Weisfeiler and A.A. Lehman. 1968. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno Technicheskaya Informatsia* 2, 9 (1968), 12–19.

[14] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2008. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res. (JAIR)* 32 (2008), 565–606.