

# Empirical Software Metrics for Benchmarking of Verification Tools

Yulia Demyanova, Thomas Pani<sup>(✉)</sup>, Helmut Veith, and Florian Zuleger

Vienna University of Technology, Vienna, Austria  
thomas.pani@tuwien.ac.at

**Abstract.** In this paper we study empirical metrics for software source code, which can predict the performance of verification tools on specific types of software. Our metrics comprise variable usage patterns, loop patterns, as well as indicators of control-flow complexity and are extracted by simple data-flow analyses. We demonstrate that our metrics are powerful enough to devise a machine-learning based portfolio solver for software verification. We show that this portfolio solver would be the (hypothetical) overall winner of both the 2014 and 2015 International Competition on Software Verification (SV-COMP). This gives strong empirical evidence for the predictive power of our metrics and demonstrates the viability of portfolio solvers for software verification.

## 1 Introduction

The success and gradual improvement of software verification tools in the last two decades is a multidisciplinary effort – modern software verifiers combine methods from a variety of overlapping fields of research including model checking, static analysis, shape analysis, SAT solving, SMT solving, abstract interpretation, termination analysis, pointer analysis etc.

The mentioned techniques all have their individual strengths, and a modern software verification tool needs to pick and choose how to combine them into a strong, stable and versatile tool. The trade-offs are based on both technical and pragmatic aspects: many tools are either optimized for specific application areas (e.g. device drivers), or towards the in-depth development of a technique for a restricted program model (e.g. termination for integer programs). Recent projects like CPA [10] and FrankenBit [20] have explicitly chosen an eclectic approach which enables them to combine different methods more easily.

There is growing awareness in the research community that the benchmarks in most research papers are only useful as proofs of concept for the individual contribution, but make comparison with other tools difficult: benchmarks are often manually selected, handcrafted, or chosen a posteriori to support a certain technical insight. Oftentimes, neither the tools nor the benchmarks are available to other researchers. The annual *International Competition on Software Verification* (SV-COMP, since 2012) [2,3,8,9] is the most ambitious attempt to remedy this situation. Now based on more than 5,500 C source files, SV-COMP

has the most diverse and comprehensive collection of benchmarks available, and is a natural starting point for a more systematic study of tool performance.

In this paper, we demonstrate that the competition results can be explained by intuitive metrics on the source code. In fact, the metrics are strong enough to enable us to construct a portfolio solver which would (hypothetically) win SV-COMP 2014 [2] and 2015 [3]. Here, a portfolio solver is a SW verification tool which uses heuristic preprocessing to select one of the existing tools [19, 24, 32].

**Table 1.** Sources of complexity for 4 tools participating in SV-COMP’15, marked with +/−/N/A when supported/not supported/no information is available. Extracted from competition reports [7] and tool papers [14, 17].

Source of complexity	CBMC	Predator	CPAchecker	SMACK	Corresp. feature
Unbounded loops	−	N/A	N/A	−	$\mathcal{L}^{\text{SB}}, \mathcal{L}^{\text{ST}}, \mathcal{L}^{\text{simple}}, \mathcal{L}^{\text{hard}}$
Pointers	+	+	+	+	PTR
Arrays	+	−	N/A	+	ARRAY_INDEX
Dynamic data structures	N/A	+	N/A	+	PTR_STRUCTURE_REC
Non-static pointer offsets	−	+	N/A	N/A	OFFSET
Non-static size of heap-allocated memory	+	+	N/A	N/A	ALLOC_SIZE
Pointers to functions	+	N/A	N/A	N/A	$m_{\text{fpcalls}}, m_{\text{fpargs}}$
Bit operations	+	−	+	−	BITVECTOR
Integer variables	+	+	+	+	SCALAR_INT
Recursion	−	−	−	+	$m_{\text{reccalls}}$
Multi-threading	+	−	−	−	THREAD_DESCR
External functions	+	−	N/A	N/A	INPUT
Structure fields	+	+	N/A	+	STRUCT_FIELD
Big CFG ( $\geq 100$ KLOC)	+	N/A	N/A	+	$m_{\text{cfgblocks}}, m_{\text{maxindeg}}$

Of course it is pointless to let a portfolio solver compete in the regular competition (except, maybe in a separate future track), but for anybody who just wants to verify software, it provides useful insights. Portfolio solvers have been successful (and controversial) in combinatorially cleaner domains such as SAT solving [25, 33, 37], quantified boolean satisfiability (QSAT) [30, 31, 34], answer set programming (ASP) [18, 27], and various constraint satisfaction problems (CSP) [19, 26, 28]. In contrast to software verification, in these areas constituent tools are usually assumed to be correct.

As an approach to software verification, portfolio solving brings interesting advantages: (1) a portfolio solver optimally uses available resources, (2) it can

avoid incorrect results of partially unsound tools, (3) machine learning in combination with portfolio solving allows us to select between multiple versions of the same tool with different runtime parameters, (4) the portfolio solver gives good insight into the state-of-the-art in software verification.

To choose the software metrics, we consider the zoo of techniques discussed above along with their target domains, our intuition as programmers, as well as the tool developer reports in their competition contributions. Table 1 summarizes these reports for tools CBMC, Predator, CPAchecker and SMACK: The first column gives obstacles the tools' authors identified, columns 2–5 show whether the feature is supported by respective tool, and the last column references the corresponding metrics, which we introduce in Sect. 2. The obtained metrics are naturally understood in three dimensions that we motivate informally first:

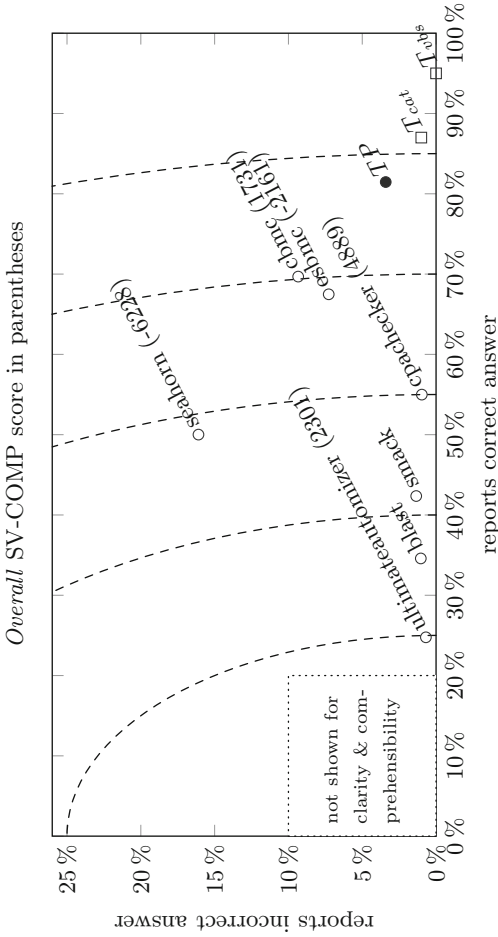
1. *Program Variables*. Does the program deal with machine or unbounded integers? Are the ints used as indices, bit-masks or in arithmetic? Dynamic data structures? Arrays? Interval analysis or predicate abstraction?
2. *Program Loops*. Reducible loops or goto programs? For-loops or ranking functions? Widening, loop acceleration, termination analysis, or loop unrolling?
3. *Control Flow*. Recursion? Function pointers? Multithreading? Simulink or complex branching?

Our hypothesis is that precise metrics along these dimensions allow us to predict tool performance. The challenge lies in identifying metrics which are predictive enough to understand the relationship between tools and benchmarks, but also simple enough to be used in a preprocessing and classification step. Sections 2.1, 2.2 and 2.3 describe metrics which correspond to the three dimensions sketched above, and are based on simple data-flow analyses.

Our algorithm for the portfolio is based on machine learning (ML) using *support vector machines* (SVMs) [12, 15] over the metrics defined above. Figure 1 depicts our experimental results on SV-COMP'15: Our tool  $\mathcal{TP}$  is the overall winner and outperforms all other tools – Sect. 4 contains a detailed discussion.

A machine-learning based method for selecting model checkers was previously introduced in [35]. Similar to our work, the authors use SVM classification with weights (cf. Sect. 3.1). Our approach is novel in the following ways:

- First, the results in [35] are not reproducible because (1) the benchmark is not publicly available, (2) the verification properties are not described, and (3) the weighting function – in our experience crucial for good predictions – is not documented.
- Second, we use a larger set of verification tools (22 tools vs. 3). Our benchmark is not restricted to device drivers and is 10 times larger (49 MLOC vs. 4 MLOC in [35]).
- Third, in contrast to structural metrics of [35] our metrics are computed using data-flow analysis. Based on tool designer reports (Table 1) we believe that they have superior predictive power. Precise comparison is difficult due to non-reproducibility of [35].



**Fig. 1.** Decisiveness-reliability plot for SV-COMP'15. The horizontal axis gives the percentage of correct answers  $c$ , the vertical axis the number of incorrect answers  $i$ . Dashed lines connect points of equal decisiveness  $c+i$ . The Overall SV-COMP score is given (if available) in parentheses.

While portfolio solvers are important, we also think that the software metrics we define in this work are interesting in their own right. Our results show that categories in SVCOMP have characteristic metrics. Thus, the metrics can be used to (1) characterize benchmarks not publicly available, (2) understand large benchmarks without manual inspection, (3) understand presence of language constructs in benchmarks.

Summarizing, in this paper we make the following contributions:

- We define software metrics along the three dimensions – program variables, program loops and control flow – in order to capture the difficulty of program analysis tasks (Sect. 2).
- We develop a machine-learning based portfolio solver for software verification that learns the best-performing tool from a training set (Sect. 3).
- We experimentally demonstrate the predictive power of our software metrics in conjunction with our portfolio solver on the software verification competitions SV-COMP’14 and SV-COMP’15 (Sect. 4).

## 2 Source Code Metrics for Software Verification

We introduce program features along the three dimensions – *program variables*, *program loops* and *control flow* – and describe how to derive corresponding metrics. Subsequent sections demonstrate their predictive power: In Sect. 3 we describe a portfolio solver for software verification based on our metrics. In Sect. 4 we experimentally demonstrate the portfolio’s success, thus attesting the descriptive and predictive power of our metrics and the portfolio.

### 2.1 Variable Role Based Metrics

The first set of features that we introduce are *variable roles*. Intuitively, a variable role is a usage pattern of how a variable is used in a program.

<pre>int n = 0, y = x; while (x) {   n++;   x = x &amp; (x-1); }</pre>	<pre>int fd = open(path, flags); int c, val=0; while (read(fd, &amp;c, 1) &gt; 0 &amp;&amp; isdigit(c)) {   val = 10*val + c-'0'; }</pre>
(a) bitvector, counter, linear	(b) character, file descriptor

**Fig. 2.** Different usage patterns of integer variables.

**Example 1.** Consider the C program in Fig. 2a, which computes the number of non-zero bits of the variable *x*. In every loop iteration, a non-zero bit of *x* is set to zero and the counter *n* is incremented. For a human reading the program, the statements *n=0* and *n++* in the loop body signal that *n* is a *counter*, and statement *x = x & (x-1)* indicates that *x* is a *bit vector*.

**Example 2.** Consider the program in Fig. 2b, which reads a decimal number from a text file and stores its numeric representation in variable `val`. Statement `fd=open(path, flags)` indicates that variable `fd` stores a *file descriptor* and statement `isdigit(c)` indicates that `c` is a *character*, because function `isdigit()` checks whether its parameter is a decimal digit character.

**Criteria for Choosing Roles.** We implemented 27 variable roles and give their informal definition in Table 2. Our choice of roles is inspired by standard concepts used by programmers. In order to create the list of roles we inspected the source code of the cBench benchmark [1] and came up with a minimum set of roles such that every variable is assigned at least one role.

**Roles as Features for Selecting a Verification Tool.** The developer reports in SV-COMP'15 [7] give evidence of the relevance of variable roles for selecting verification tools. Most often authors mention language constructs which – depending on whether they are fully, partially, or not modeled by a tool – constitute its strong or weak points. We give examples of such constructs in Table 1 and relate them to variable roles. A preliminary experiment in [16], where we have successfully used variable roles to predict categories in SV-COMP'13, gives further evidence for our claim.

**Definition of Roles.** We define roles using data-flow analysis, an efficient fixed-point algorithm popular in optimizing compilers [6]. Our current definition of roles is control-flow insensitive, and the result of analysis is a set of variables  $Res^R$  which are assigned role  $R$ . We give the definition of variable roles in [16].

**Example 3.** We describe the process of computing roles on the example of role LINEAR for the code in Fig. 2a. Initially, the algorithm assigns to  $Res^{\text{LINEAR}}$  the set of all variables  $\{x, y, n\}$ . Then it computes the greatest fixed point in three iterations. In iteration 1, variable `x` is removed, because it is assigned expression `x & (x-1)`, resulting in  $Res^{\text{LINEAR}} = \{y, n\}$ . In iteration 2, variable `y` is removed, because it is assigned variable `x`, resulting in  $Res^{\text{LINEAR}} = \{n\}$ . In iteration 3,  $Res^{\text{LINEAR}}$  does not change, and the result of the analysis is  $Res^{\text{LINEAR}} = \{n\}$ .

**Definition 1 (Variable Role Based Metrics).** For a given benchmark file  $f$ , we compute the mapping  $Res^R$  from variable roles to the program variables of  $f$ . We derive role metrics  $m_R$  that represent the relative occurrence of each variable role  $R$ :  $m_R = |Res^R|/|Vars|$ , where  $R \in Roles$ .

## 2.2 Loop Pattern Based Metrics

The second set of program features we introduce is a classification of loops. The capability of Turing complete imperative languages to express unbounded

**Table 2.** List of variable roles with informal definitions. Type *struct\_type* stands for a C structure, *any\_type* for an arbitrary C type.

C type	Role name	Informal definition
int	ARRAY_INDEX	Occurs in an array subscript expression
	ALLOC_SIZE	Passed to a standard memory allocation function
	BITVECTOR	Used in a bitwise operation or assigned the result of a bitwise operation or a BITVECTOR variable
	BOOL	Assigned and compared only to 0,1, the result of a boolean operation or a BOOL variable
	BRANCH_COND	Used in the condition of an if statement
	CHAR	Used in a library function which manipulates characters, or assigned a character literal
	CONST_ASSIGN	Assigned only literals or CONST_ASSIGN variables
	COUNTER	Changed only in increment/decrement statements
	FILE_DESCR	Passed to a library function which manipulates files
	INPUT	Assigned the result of an external function or passed to it as a parameter by reference
	LINEAR	Assigned only linear combinations of LINEAR variables
	LOOP_BOUND	Used in a loop condition in a comparison operation, where it is compared to a LOOP_ITERATOR variable
	LOOP_ITERATOR	Occurs in loop condition, assigned in loop body
	MODE	Not used in comparison operations other than == and !=; assigned and compared to constant values only
	OFFSET	Added to or subtracted from a pointer
	SCALAR_INT	Scalar integer variable
	SYNT_CONST	Not assigned in the program (a global or an unused variable, or a formal parameter to a global function)
	THREAD_DESCR	Passed to a function of pthread library
USED_IN_ARITHM	Used in addition/subtraction/multiplication/division	
float	SCALAR_FLOAT	Scalar float variable
int*, float*	PTR_SCALAR	Pointer to a scalar value
<i>struct_type</i> *	PTR_STRUCT	Pointer to a structure
	PTR_STRUCT_PTR	Pointer to a structure which has a pointer field
	PTR_STRUCT_REC	Pointer to a recursively defined structure
	PTR_COMPL_STRUCT	Pointer to a recursively defined structure with more than one pointer, e.g. doubly linked lists
<i>any_type</i> *	HEAP_PTR	Assigned the result of a memory allocation
	PTR	Pointer to any value

iteration entails hard and in general undecidable problems for any non-trivial program analysis. On the other hand, in many cases iteration takes trivial forms, for example in loops enumerating a bounded range (counting). In [29] we intro-

duce a family of *loop patterns* that capture such differences. Ability to reason about bounds or termination of loops allows a verification tool to discharge the (un)reachability of assertions after the loop, or to compute unrolling factors and soundness limits in the case of bounded model checking. Thus we expect our loop patterns to be useful program features for constructing our portfolio.

**Criteria for Choosing Loop Patterns.** We start with a termination procedure for a restricted set of bounded loops. This loop pattern is inspired by basic (bounded) FOR-loops, a frequently used programming pattern. It allows us to implement an efficient termination procedure using syntactic pattern matching and data-flow analysis. Additionally, this loop class lends itself to derive both a stronger notion of boundedness, and weaker notions (heuristics) of termination. We give an informal description of these patterns in Table 3; for details cf. [29]

**Usefulness of Loop Patterns.** In [29] we give evidence that these loop patterns are a common engineering pattern allowing us to describe loops in a variety of benchmarks, that they indeed capture classes of different empirical hardness, and that the hardness increases as informally described in Table 3.

**Definition 2 (Loop Pattern Based Metrics).** *For a given benchmark file  $f$ , we compute  $\mathcal{L}^{\text{SB}}$ ,  $\mathcal{L}^{\text{ST}}$ ,  $\mathcal{L}^{\text{simple}}$ ,  $\mathcal{L}^{\text{hard}}$ , and the set of all loops  $\text{Loops}$ . We derive loop metrics  $m_P$  that represent the relative occurrence of each loop pattern  $P$ :  $m_P = |\mathcal{L}^P|/|\text{Loops}|$  where  $P \in \{\text{ST}, \text{SB}, \text{simple}, \text{hard}\}$ .*

**Table 3.** List of loop patterns with informal descriptions.

Loop pattern	Empirical hardness	Informal definition
Syntactically bounded loops $\mathcal{L}^{\text{SB}}$	Easy	The number of executions of the loop body is bounded (considers outer control flow)
Syntactically terminating loops $\mathcal{L}^{\text{ST}}$	Intermediate	The loop terminates whenever control flow enters it (disregards outer control flow)
Simple loops $\mathcal{L}^{\text{simple}}$	Advanced	A heuristic derived from syntactically terminating loops by weakening the termination criteria. A good heuristic for termination
Hard loops $\mathcal{L}^{\text{hard}}$	Hard	Any loop that is not classified as simple



### 2.3 Control Flow Based Metrics

Complex control flow poses another challenge for program analysis. To measure its presence, we introduce five additional metrics: For *control flow complexity*, we count (a) the number of basic blocks in the control flow graph (CFG)  $m_{\text{cfgblocks}}$ , and (b) the maximum indegree of any basic block in the CFG  $m_{\text{maxindeg}}$ . To represent the use of *function pointers*, we measure (a) the ratio of call expressions taking a function pointer as argument  $m_{\text{fpcalls}}$ , and (b) the ratio of function call arguments that have a function pointer type  $m_{\text{fpargs}}$ . Finally, to describe the use of *recursion*, we measure the number of direct recursive function calls  $m_{\text{recalls}}$ .

## 3 A Portfolio Solver for Software Verification

### 3.1 Preliminaries on Machine Learning

In this section we introduce standard terminology from the machine learning (ML) community as can for example be found in [11].

**Data Representation.** A *feature vector* is a vector of real numbers  $\mathbf{x} \in \mathbb{R}^n$ . A *labeling function*  $L : X \rightarrow Y$  maps a set of feature vectors  $X \subseteq \mathbb{R}^n$  to a set  $Y \subseteq \mathbb{R}$ , whose elements are called *labels*.

**Supervised Machine Learning.** In *supervised* ML problems, labeling function  $L$  is given as input. *Regression* is a supervised ML problem where labels are real numbers  $Y \subseteq \mathbb{R}$ . In *classification*, in contrast, labels belong to a finite set of integers  $Y \subseteq \mathbb{Z}$ . *Binary classification* considers two classes  $Y = \{1, -1\}$ , and a problem with more than two classes is called *multi-class classification*.

Given a set of feature vectors  $X$ , labeling function  $L$  and error measure function  $Err : \mathbb{R}^s \times \mathbb{R}^s \rightarrow \mathbb{R}$ , where  $s = |X|$ , a supervised ML algorithm searches for function  $M : \mathbb{R}^n \rightarrow Y$  in some function space such that the value  $Err(L(X), M(X))$  is minimal.

**Support Vector Machine.** A *support vector machine* (SVM) [12,15] is a supervised ML algorithm, parametrized by a *kernel function*  $K(x_i, x_j) \equiv \phi(x_i)^T \phi(x_j)$ , that finds a hyperplane  $w\phi(x_i) - b = 0$  separating the data with different labels. In the case of binary classification,

$$M(x) = \text{sign} \left( \sum_{i=1}^s w_i L(x_i) \phi(x_i)^T \phi(x) \right) \text{ and } Err = \frac{1}{2} w^T w + C \sum_{i=1}^s \xi_i \quad (1)$$

where  $\text{sign}(n) = \begin{cases} -1 & \text{if } n < 0 \\ 1 & \text{if } n \geq 0 \end{cases}$ ,  $x_i \in X$  is a feature vector,  $C > 0$  is the penalty parameter of the error term, function  $\phi$  is implicitly given through kernel function

$K$ , and  $w$ ,  $b$  and  $\xi$  are existentially quantified parameters of the optimization problem

$$\min_{w,b,\xi} Err, \text{ subject to } L(x_i)(w^T \phi(x_i) + b) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \quad (2)$$

with  $\xi_i$  measuring the degree of misclassification of point  $x_i$ .

The kernel function  $K$  and  $C \in \mathbb{R}$  are parameters of SVM. An example of a non-linear kernel function is the *Radial Basis Function* (RBF):  $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$ ,  $\gamma > 0$ .

**Probabilistic Classification.** *Probabilistic classification* is a generalization of the classification algorithm, which searches for a function  $M^P : \mathbb{R}^n \rightarrow (Y \rightarrow [0, 1])$  mapping a feature vector to a *class probability* distribution, which is a function  $P : Y \rightarrow [0, 1]$  from a set of classes  $Y$  to the unit interval. There is a standard algorithm for estimating class probabilities for SVM [36].

**Creating and Evaluating a Model.** Function  $M$  is called a *model*, the set  $X$  used for creating the model is called *training set*, and the set used for evaluating the model  $X'$  is called *test set*.

To avoid overly optimistic evaluation of the model, it is common to require that the training and test sets are disjoint:  $X \cap X' = \emptyset$ . A model which produces accurate results with respect to the error measure for the training set, but results in a high error for previously unseen feature vectors  $x \notin X$ , is said to *overfit*.

**Data Imbalances.** Labeling function  $L$  is said to be *imbalanced* when it exhibits an unequal distribution between its classes:  $\exists y_i, y_j \in Y. Num(y_i) / Num(y_j) \sim 100$ , where  $Num(y) = |\{x \in X \mid L(x) = y\}|$ , i.e. imbalances of the order 100:1 and higher. Data imbalances significantly compromise the performance of most standard learning algorithms [21].

A common solution for the imbalanced data problem is to use a *weighting function*  $W : X \rightarrow \mathbb{R}$  [23]. *SVM with weights* is a generalization of SVM, where  $Err = \frac{1}{2} w^T w + C \sum_{i=1}^s W(x_i) \xi_i$ .  $W$  is usually chosen empirically.

An orthogonal solution of dealing with data imbalances is the reduction of a multi-class classification problem to multiple binary classification problems: *one-vs-all* classification creates one model per class  $i$ , with the labeling function  $L_i(x) = \begin{cases} 1 & \text{if } L(x) = i \\ -1 & \text{otherwise} \end{cases}$ , and the predicted value calculated as  $M(x) = \text{choose}(\{i \mid M_i(x) = 1\})$ , where a suitable operator *choose* is used to choose a single class from multiple predicted classes.

### 3.2 The Competition on Software Verification SV-COMP

**Setup.** A verification task in SV-COMP is given as a C source file  $f$  and a verification property  $p$ . The property is either a label reachability check or a memory safety check (comprising checks for freedom of unsafe deallocations, unsafe pointer dereferences, and memory leaks). The expected answer  $ExpAns$  is provided for each task by the designers of the benchmark. The verification tasks are partitioned into *categories*, manually grouped by characteristic features such as usage of bitvectors, concurrent programs, linux device drivers, etc.

**Scoring.** The competition assigns a *score* to each tool’s result on a verification task  $v$ . The *category score* of a tool is defined as the sum of scores for individual tasks in the category. In addition, medals are awarded to the three best tools in each category. The *Overall SV-COMP score* considers all verification tasks, with each constituent category score normalized by the number of tasks in it.

### 3.3 Tool Selection as a Machine Learning Problem

In this section, we first describe the setup of our portfolio solver  $\mathcal{TP}$ , and then define the notion of the *best-performing tool*  $t^{best}$  predicted by  $\mathcal{TP}$ .

**Definitions.** A *verification task*  $v = \langle f, p, type \rangle$  is a triple of a source file  $f$ , the property  $p$  and property type  $type$  (e.g. reachability or safety). Function  $ExpAns : Tasks \rightarrow \{true, false\}$  maps verification task  $v \in Tasks$  to *true* if the property  $p$  holds for  $f$  and to *false* otherwise. We identify each *verification tool* by a unique natural number  $t \in \mathbb{N}$ .

The result of a run of a tool  $t$  on a verification task  $v$  is a pair  $\langle ans_{t,v}, runtime_{t,v} \rangle$ , where  $ans_{t,v} \in \{true, false, unknown\}$  is the tool’s answer whether the property holds, and  $runtime_{t,v} \in \mathbb{R}$  is the runtime of the tool in seconds. The *expected answer* for a task  $v$  is a boolean value  $ExpAns(v)$ .

**Machine Learning Data.** We compute feature vectors from the metrics and the results of the competition as follows: for verification task  $v$  we define feature vector  $\mathbf{x}(v) = (m_{ARRAY\_INDEX}(v), \dots, m_{PTR}(v), m_{ST}(v), \dots, m_{hard}(v), m_{cfgblocks}(v), \dots, m_{reccalls}(v), type(v))$ , where the  $m_i(v)$  are our metrics from Sect. 2 and  $type(v) \in \{0, 1\}$  encodes if the property is reachability or memory safety.

The portfolio solver predicts a tool identifier  $t \in \{1, \dots, n\}$ , which is a multi-class classification problem. We use a generalization of the one-vs-all classification to solve the problem. We define the labeling function  $L_t(v)$  for tool  $t$  and task  $v$  as follows:

$$L_t(v) = \begin{cases} 1 & \text{if } ans_{t,v} = ExpAns(v) \\ 2 & \text{if } ans_{t,v} = unknown \\ 3 & \text{if } ans_{t,v} \neq unknown \wedge ans_{t,v} \neq ExpAns(v) \end{cases}$$

where we treat opted-out categories as if the tool answered *unknown* for all of the category’s verification tasks.

**Formulation of the Machine Learning Problem.** Given  $|Tools|$  classification problems for a task  $v$ , the portfolio algorithm chooses a tool  $t^{best}$  as follows:

$$t^{best} = \begin{cases} choose(TCorr(v)) & \text{if } TCorr(v) \neq \emptyset \\ choose(TUnk(v)) & \text{if } TCorr(v) = \emptyset \wedge TUnk(v) \neq \emptyset \\ t^{winner} & \text{if } TCorr(v) = \emptyset \wedge TUnk(v) = \emptyset \end{cases}$$

where  $TCorr(v) = \{t \in Tools \mid M_t(v) = 1\}$ ,  $TUnk(v) = \{t \in Tools \mid M_t(v) = 2\}$  and  $t^{winner}$  is the winner of the competition, e.g. CPAchecker in SV-COMP’15. We now describe two alternative ways of implementing the operator *choose*.

1. **“Success/Fail + Time”:**  $\mathcal{TP}^{SuccFailTime}$ . We formulate  $|Tools|$  additional regression problems, where the predicted value is the runtime of the tool  $runtime_{t,v}$ . We define  $choose(T) = \arg \min_{t \in T} runtime_{t,v}$ .
2. **“Success/Fail + Probability”:**  $\mathcal{TP}^{SuccFailProb}$ . We define the operator  $choose(T) = \arg \max_{t \in T} P_{t,v}$ , where  $P_{t,v}$  is class probability estimate.

In Table 4 we compare the two *choose* operators for category *Overall* in the setup of SV-COMP’14 according to 3 different criteria: the score, the percentage of correctly and incorrectly answered tasks and the place in the competition.

Configuration  $\mathcal{TP}^{SuccFailProb}$  yields a higher score and number of correct answers with less runtime. We believe this is due to the tool runtimes varying in the range of 5 orders of magnitude (from tenth parts of a second to 15 min), which causes high error rates in the predicted runtime. We therefore use configuration  $\mathcal{TP}^{SuccFailProb}$  and in the following refer to it as  $\mathcal{TP}$ .

**Table 4.** Comparison of 2 formulations of  $\mathcal{TP}$ .

Setting	Correct/Incorrect/ Unknown answers, %	Score	Runtime, s	Place
$\mathcal{TP}^{SuccFailTime}$	92/3/6	1384	279859	1
$\mathcal{TP}^{SuccFailProb}$	93/1/5	1494	132688	1

**The Weighting Function.** We analyzed the results of SV-COMP’14 and observed, that the labeling function in the formulation of  $\mathcal{TP}^{SuccFailProb}$  is highly imbalanced: the label which corresponds to incorrect answers,  $L_t(v) = 3$ , occurs in less than 4% for all tools.

We therefore use SVM with weights, in accordance with the standard practice in machine learning. We note that we use the same weighting function for our

experiments in the setup of SV-COMP'15 without any changes. Given a task  $v$  and tool  $t$ , we calculate the weighting function  $W$  as follows:

$$W(v, t) = Potential(v) * Criticality(v) * Performance(t, Cat(v)) * Speed(t, Cat(v))$$

- where  $Potential(v) = score_{max}(v) - score_{min}(v)$  is the difference of the maximal and minimal possible scores for task  $v$ . For example, in the setup of SV-COMP'14, if  $v$  is safe, then  $score_{max}(v) = 2$  and  $score_{min}(v) = -8$ ;
- $Criticality(v) = \frac{1}{|\{t \in Tools \mid ans_{t,v} = ExpAns(v)\}|}$  is inversely proportional (subject to a constant factor) to the probability of randomly choosing a tool which gives the expected answer;
- $Performance(t, c) = \frac{cat\_score(t, c)}{cat\_score(t^{cbest}, c)}$  is the ratio of the scores of tool  $t$  and the best in category  $c$  tool  $t^{cbest}$ , where given the score  $score_{t,v}$  of tool  $t$  for task  $v$ ,  $t^{cbest} = \arg \max_{t_i \in Tools} (cat\_score(t_i, c))$  and  $cat\_score(t, c) = \sum_{\{v \in Tasks \mid Cat(v)=c\}} (score_{t,v})$ ;
- $Speed(t, c) = \frac{\ln(rel\_time(t, c))}{\ln(rel\_time(t^{cfst}, c))}$  is the relative difference of the orders of magnitude of the fraction in total runtime of the time spent by tool  $t$  and the fastest in category  $c$  tool  $t^{cfst}$  respectively, where  $rel\_time(t, c) = (cat\_time(t, c)) / \left( \sum_{t_i \in Tools} cat\_time(t_i, c) \right)$ ,  $t^{cfst} = \arg \min_{t_i \in Tools} (cat\_time(t_i, c))$  and  $cat\_time(t, c) = \sum_{\{v \in Tasks \mid Cat(v)=c\}} runtime_{t,v}$ .

**Implementation of  $\mathcal{TP}$ .** Finally, we discuss the details of the implementation of  $\mathcal{TP}$ . We use the SVM ML algorithm with the RBF kernel and weights implemented in the LIBSVM library [13]. To find optimal parameters for a ML algorithm with respect to the error measure function, we do exhaustive search on the grid, as described in [22].

## 4 Experimental Results

### 4.1 SV-COMP 2014 vs. 2015

As described in Sect. 3.2, SV-COMP provides two metrics for comparing tools: score and medal counts. As the scoring policy has recently changed (the penalties for incorrect answers were increased) after a close jury vote [4], we are interested in how stable the scores are under different scoring policies. The following table gives the three top-scoring tools in *Overall* and their scores in SV-COMP'14 and '15, as well as the top-scorers of SV-COMP'14 if the 2015 scoring policy had been applied, and vice versa:

Competition	Scoring	1 <sup>st</sup> place (score)	2 <sup>nd</sup> place (score)	3 <sup>rd</sup> place (score)
SV-COMP'14	Original	CBMC (3,501)	CPAchecker (2,987)	LLBMC (1,843)
	Like '15	CPAchecker (3,035)	CBMC (2,515)	LLBMC (2,004)
SV-COMP'15	Original	CPAchecker (4,889)	Ult. Aut. (2,301)	CBMC (1,731)
	Like '14	CPAchecker (5,537)	SMACK (4,120)	CBMC (3,481)

*Discussion.* Clearly, the scoring policy has a major impact on the competition results: If the '15 policy is applied to SV-COMP'14, the first and second placed tools switch ranks. SV-COMP'15, applying the previous year's policy has an even stronger effect: Ultimate Automizer loses its silver medal to SMACK, a tool originally not among the top three, and CBMC almost doubles its points.

Given that SV-COMP score and thus also medal counts are rather volatile, we introduce *decisiveness-reliability plots* (DR-plots) in the next section to complement our interpretation of the competition results.

## 4.2 Decisiveness-Reliability Plots

To better understand the competition results, we create scatter plots where each data point  $\mathbf{v} = (c, i)$  represents a tool that gives  $c\%$  correct answers and  $i\%$  incorrect answers. Figures 1 and 3 show such plots based on the verification tasks in SV-COMP'14 and '15. Each data point marked by an unfilled circle  $\circ$  represents one competing tool. The rectilinear distance  $c + i$  from the origin gives a tool's *decisiveness*, i.e. the farther from the origin, the fewer times a tool reports "unknown". The angle enclosed by the horizontal axis and  $\mathbf{v}$  gives a tool's *(un)reliability*, i.e. the wider the angle, the more often the tool gives incorrect answers. Thus, we call such plots *decisiveness-reliability plots* (DR-plots).

*Discussion.* Figures 1 and 3 show DR-plots for the verification tasks in SV-COMP'14 and '15. For 2014, all the tools are performing quite well on soundness: none of them gives more than 4% of incorrect answers. CPAchecker, ESBMC and CBMC are highly decisive tools, with more than 83% correct answers.

In 2015 (Fig. 1) the number of verification tasks more than doubled, and there is more variety in the results: We see that very reliable tools (BLAST, SMACK, and CPAchecker) are limited in decisiveness – they report "unknown" in more than 40% of cases. The bounded model checkers CBMC and ESBMC are more decisive at the cost of giving up to 10% incorrect answers. We also give *Overall* SV-COMP scores (where applicable) in parentheses. Clearly, tools close together in the DR-plot not necessarily have similar scores because of the different score weights prescribed by the SV-COMP scoring policy.

Referring back to Figs. 1 and 3, we also show the theoretic strategies  $T_{cat}$  and  $T_{vbs}$  marked by a square  $\square$ : Given a verification task  $v$ ,  $T_{cat}$  selects the tool winning the corresponding competition category  $Cat(v)$ .  $T_{vbs}$  is the *virtual best solver* (VBS) and selects the best performing tool per verification

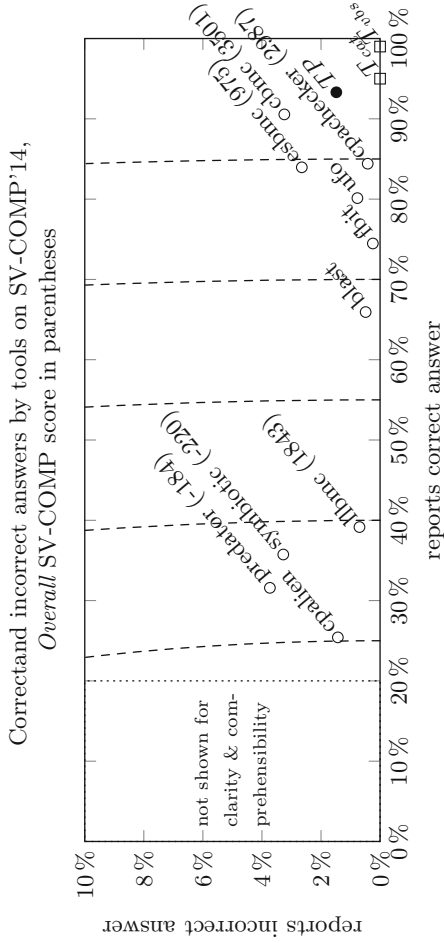


Fig. 3. Decisiveness-reliability plot for SV-COMP'14.

(a) Overall SV-COMP score, runtime and medal counts for SV-COMP'14.

	blast	cbmc	cpa- che- cker	cpa- lien	esbmc	fbit	llbmc	ufo	$\mathcal{TP}$	$T_{cat}$	$T_{vbs}$
<i>Overall</i>	468	<b>1292</b>	<b>1235</b>	266	695	666	853	735	<b>1494</b>	1732	1840
	2066	<b>4991</b>	<b>1865</b>	776	4024	898	978	381	<b>2211</b>	1310	270
Medals	1/0/0	2/2/2	2/1/1	0/0/0	1/0/1	0/0/2	1/0/1	1/1/0	1/5/1	-	-

(b) Overall SV-COMP score, runtime and medal counts for SV-COMP'15.

	blast	cas- cade	cbmc	cpa- che- cker	pre- da- torhp	smack	ulti- mate- kojak	ulcseq	$\mathcal{TP}$	$T_{cat}$	$T_{vbs}$
<i>Overall</i>	737	806	684	<b>2228</b>	389	<b>1542</b>	1215	273	<b>2511</b>	3231	3768
	4546	5146	11936	<b>6288</b>	96	<b>8727</b>	7979	12563	<b>6260</b>	4360	1882
Medals	1/0/0	0/0/0	1/1/1	2/1/5	1/0/1	2/1/1	0/2/0	0/0/0	1/6/1	-	-

**Fig. 4.** Experimental results for the eight best competition participants in *Overall*, plus our portfolio  $\mathcal{TP}$  and the idealized strategies  $T_{cat}$ ,  $T_{vbs}$  on *random subsets* of SV-COMP, given as arithmetic mean of 10 experiments on the resp. test sets  $test_{year}$ . The first row shows the *Overall* SV-COMP score and beneath it the runtime in minutes. We highlight the gold, silver, and bronze medal in dark gray, light gray and white+bold font, respectively. The second row shows the number of gold/silver/bronze medals won in individual categories.

task. Both strategies illustrate that combining tools can yield an almost perfect solver, with  $\geq 95\%$  correct and  $0\%$  incorrect answers. (Note that these figures may give an overly optimistic picture – after all the benchmarks are supplied by the competition participants.) The results for  $T_{vbs}$  compared to  $T_{cat}$  indicate that leveraging not just the category winner provides an advantage in both reliability and decisiveness. A useful portfolio would thus lie somewhere between CPAchecker, CBMC,  $T_{cat}$ , and  $T_{vbs}$ , i.e. improve upon the decisiveness of constituent tools while minimizing the number of incorrect answers.

### 4.3 Evaluation of Our Portfolio Solver

We implemented the ML-based portfolio  $\mathcal{TP}$  for SV-COMP'14 in our tool *Verifolio* [5]. When competition results for SV-COMP'15 became available, we successfully evaluated the existing techniques on the new data. We present these results both in terms of the traditional metrics used by the competition (SV-COMP score and medals), and by its placement in DR-plots:

*Setup.* For our experiments we did not rebuild the infrastructure of SV-COMP, but use numeric results from the competition to compare our portfolio approach against other tools. Following a standard practice in ML [11], we randomly split the verification tasks of SV-COMP'*year* into a training set  $train_{year}$  and a test set  $test_{year}$  with a ratio of 60:40. We train our portfolio on  $train_{year}$  and re-run the competition on  $test_{year}$ , with the portfolio participating as an additional



tool. As the partitioning into training and test sets is randomized, we conduct the experiment 10 times and report the arithmetic mean of all figures. Figures 4a and b show the *Overall* SV-COMP scores, runtimes and medal counts. The DR-plots in Figs. 1 and 3 show the portfolio marked by a filled circle •.

*Overhead of Feature Extraction.* By construction, our portfolio incurs an overhead for feature extraction and prediction before actually executing the selected tool. We find this overhead to be negligible with a median time of  $\tilde{x}_{\text{features}} = 0.5$  seconds for feature extraction and  $\tilde{x}_{\text{prediction}} = 0.5$  seconds for prediction.

*Discussion.* First, we discuss our results in terms of *Overall* SV-COMP score and medals. The experimental results for SV-COMP’14 in Fig. 4a show that our portfolio overtakes the original *Overall* winner CBMC with 16 % more points. It wins a total of seven medals (1/5/1 gold/silver/bronze) compared to CBMC’s six medals (2/2/2). For SV-COMP’15 (Fig. 4b), our portfolio  $\mathcal{TP}$  is again the strongest tool, collecting 13 % more points than the original *Overall* winner CPAchecker. Both CPAchecker and  $\mathcal{TP}$  collect 8 medals, with CPAchecker’s 2/1/5 against  $\mathcal{TP}$ ’s 1/6/1.

Second, we discuss the DR-plots in Figs. 1 and 3. Our portfolio  $\mathcal{TP}$  positions itself between CPAchecker, CBMC and the theoretic strategies  $T_{\text{cat}}$  and  $T_{\text{vbs}}$ . Furthermore,  $\mathcal{TP}$  falls halfway between the concrete tools and idealized strategies. We think this is a promising result, but also leaves room for future work. Here we invite the community to contribute further feature definitions, learning techniques, portfolio setups, etc. to enhance this approach.

*Constituent Verifiers Employed by Our Portfolio.* Our results could suggest that  $\mathcal{TP}$  implements a trade-off between CPAchecker’s conservative-and-sound and CBMC’s decisive-but-sometimes-unsound approach. Contrarily, our experiments show that significantly more tools get selected by our portfolio solver. Additionally, we find that our approach is able to select domain-specific solvers: For example, in the *Concurrency* category,  $\mathcal{TP}$  almost exclusively selects variants of CSeq, which translates concurrent programs into equivalent sequential ones.

*Wrong Predictions.* Finally, we investigate cases of wrong predictions made by the portfolio solver, which are due to two reasons:

First, ML operates on the assumption that the behavior of a verification tool is the same for different verification tasks with the same or very similar metrics. However, sometimes this is not the case because tools are (1) unsound, e.g. SMACK in category Arrays, (2) buggy, e.g. BLAST in DeviceDrivers64, or (3) incomplete, e.g. CPAchecker in ECA.

Second, the data imbalances lead to the following bias in ML: For a verification tool that is correct most of the time, ML will prefer the error of predicting that the tool is correct (when in fact incorrect) over the error that a tool is incorrect (when in fact correct), i.e. “good” tools are predicted to be even “better”.

## References

1. Collective benchmark (cBench). <http://ctuning.org/wiki/index.php/CTools:CBench>. Accessed 6 Feb 2015
2. Competition on Software Verification 2014. <http://sv-comp.sosy-lab.org/2014/>. Accessed 6 Feb 2015
3. Competition on Software Verification 2015. <http://sv-comp.sosy-lab.org/2015/>. Accessed 6 Feb 2015
4. SV-COMP 2014 - Minutes. <http://sv-comp.sosy-lab.org/2015/Minutes-2014.txt>. Accessed 6 Feb 2015
5. Verifolio. <http://forsyte.at/software/verifolio/>. Accessed 11 May 2015
6. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading (1986)
7. Baier, C., Tinelli, C. (eds.): *TACAS 2015*. LNCS, vol. 9035. Springer, Heidelberg (2015)
8. Beyer, D.: Status report on software verification. In: Ábrahám, E., Havelund, K. (eds.) *TACAS 2014 (ETAPS)*. LNCS, vol. 8413, pp. 373–388. Springer, Heidelberg (2014)
9. Beyer, D.: Software verification and verifiable witnesses. In: Baier, C., Tinelli, C. (eds.) *TACAS 2015*. LNCS, vol. 9035, pp. 401–416. Springer, Heidelberg (2015)
10. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
11. Bishop, C.M.: *Pattern Recognition and Machine Learning*. Springer, New York (2006)
12. Boser, B.E., Guyon, I., Vapnik, V.: A training algorithm for optimal margin classifiers. In: *Conference on Computational Learning Theory (COLT 1992)*, pp. 144–152 (1992)
13. Chang, C., Lin, C.: LIBSVM: a library for support vector machines. *ACM TIST* **2**(3), 27 (2011)
14. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
15. Cortes, C., Vapnik, V.: Support-vector networks. *Mach. Learn.* **20**(3), 273–297 (1995)
16. Demyanova, Y., Veith, H., Zuleger, F.: On the concept of variable roles and its use in software analysis. In: *Formal Methods in Computer-Aided Design (FMCAD 2013)*, pp. 226–230 (2013)
17. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Logozzo, F., Fähndrich, M. (eds.) *Static Analysis*. LNCS, vol. 7935, pp. 215–237. Springer, Heidelberg (2013)
18. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M.T., Ziller, S.: A portfolio solver for answer set programming: preliminary report. In: Delgrande, J.P., Faber, W. (eds.) *LPNMR 2011*. LNCS, vol. 6645, pp. 352–357. Springer, Heidelberg (2011)
19. Gomes, C.P., Selman, B.: Algorithm portfolios. *Artif. Intell.* **126**(1–2), 43–62 (2001)
20. Gurfinkel, A., Belov, A.: FRANKENBIT: bit-precise verification with many bits. In: Ábrahám, E., Havelund, K. (eds.) *TACAS 2014 (ETAPS)*. LNCS, vol. 8413, pp. 408–411. Springer, Heidelberg (2014)

21. He, H., Garcia, E.A.: Learning from imbalanced data. *Knowl. Data Eng.* **21**(9), 1263–1284 (2009)
22. Hsu, C.W., Chang, C.C., Lin, C.J., et al.: A practical guide to support vector classification (2003)
23. Huang, Y.M., Du, S.X.: Weighted support vector machine for classification with uneven training class sizes. *Mach. Learn. Cybern.* **7**, 4365–4369 (2005)
24. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. *Science* **275**(5296), 51–54 (1997)
25. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm selection and scheduling. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 454–469. Springer, Heidelberg (2011)
26. Lobjois, L., Lemaître, M.: Branch and bound algorithm selection by performance prediction. In: Mostow, J., Rich, C. (eds.) National Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference, pp. 353–358 (1998)
27. Maratea, M., Pulina, L., Ricca, F.: The multi-engine ASP solver ME-ASP. In: del Cerro, L.F., Herzig, A., Mengin, J. (eds.) JELIA 2012. LNCS, vol. 7519, pp. 484–487. Springer, Heidelberg (2012)
28. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., OSullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish Conference on Artificial Intelligence and Cognitive Science (2008)
29. Pani, T.: Loop patterns in C programs. Diploma Thesis (2014). <http://forsyte.at/static/people/pani/sloopy/thesis.pdf>
30. Pulina, L., Tacchella, A.: A multi-engine solver for quantified boolean formulas. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 574–589. Springer, Heidelberg (2007)
31. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints* **14**(1), 80–116 (2009)
32. Rice, J.R.: The algorithm selection problem. *Adv. Comput.* **15**, 65–118 (1976)
33. Roussel, O.: Description of pfolio. <http://www.cril.univ-artois.fr/~roussel/pfolio/solver1.pdf>
34. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: Proceedings of the Conference on Artificial Intelligence (AAAI), pp. 255–260 (2007)
35. Tulsian, V., Kanade, A., Kumar, R., Lal, A., Nori, A.V.: MUX: algorithm selection for software model checkers. In: Working Conference on Mining Software Repositories, pp. 132–141 (2014)
36. Wu, T.F., Lin, C.J., Weng, R.C.: Probability estimates for multi-class classification by pairwise coupling. *J. Mach. Learn. Res.* **5**, 975–1005 (2004)
37. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. (JAIR)* **32**, 565–606 (2008)