Targeted Configuration of an SMT Solver*

Jan Hůla, Jan Jakubův, Mikoláš Janota, and Lukáš Kubej

Czech Technical University in Prague, Prague, Czech Republic

Abstract. We present a generic method to configure an automated reasoning solver in order to increase its performance on selected target problems. We describe a strategy invention system Grackle that is designed to invent a set of strong and complementary solver strategies. The strategies are then used to train a gradient boosted decision tree model to select the best strategy for a specific input problem. We evaluate our method on the SMT solver Bitwuzla and we obtain a significant increase in the number of solved problems, and a substantial decrease in runtime.

Keywords: Satisfiability Module Theories · Strategy Invention · Strategy Scheduling · Machine Learning

1 Introduction

Automated reasoning solvers, such as automated theorem provers (ATPs), satisfiability modulo theories (SMT) solvers, and SAT solvers, are important tools when dealing with computer mathematics. Formal mathematics and interactive theorem provers (ITPs), such as Mizar [15], Coq [12], and Isabelle [25], are making increasing use of automated reasoning solvers, for example, in the form of dedicated systems called hammers [9]. This involves translating the problems into the language understood by the underlying solver. Problems coming from such translations are quite often different from problems that the solver is optimized for. Hence, optimization of solvers to specific target problems is becoming a topic of increasing importance.

Many solvers allow a user to specify various options to influence the inner workings of the solver. These options are quite often the only way for the user to target the solver to specific problems. By a solver *strategy*, we understand a collection of solver options and their values, which are used to influence the behavior of the solver. In this paper, we consider the important question of an automated configuration of a solver to user-specified problems by deployment of targeted strategies.

Our method is a combination of *strategy invention* and *strategy selection*. In Section 2 and Section 3, we present a generic strategy invention system Grackle,

* This work was supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project *POSTMAN* no. LL1902, and by the ERC Starting grant *SMART* no. 714034. This scientific article is part of the *RICAIP* project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

designed to invent a set of strong and complementary strategies for an arbitrary parametrized solver. Section 4 then describes a method for selecting the best strategy for a given problem. The strategy selection is implemented by gradient boosted decision trees, and it tries to predict the best strategy based on syntactic features of the problem.

The new system Grackle, a successor of the system called BliStr [31], is a strategy invention system for a generic solver. It is designed to invent a portfolio (collection) of well-performing but complementary strategies targeted to user-specified input problems. In this way, the users can develop their own sets of strategies for specific problems, and does not need to rely on the strategies predefined in the solver. As opposed to Grackle, the ancestor BliStr is hardwired to the first-order logic theorem prover E and does not combine strategy invention with different strategy selection modes.

Grackle is based on an evolutionary algorithm, where strategies are considered “animals”, and problems to be solved are considered their “food”. Only the animals that consume enough food, that is, solve enough problems, survive to the next generation and are given the chance to conceive an offspring. This algorithm favors animals that consume food that is not consumed by others. This leads to the diversity and complementarity of the invented strategies. The name *Grackle* is motivated by the common name of passerine birds native to North and South America. During evolution, different grackle species developed different bill sizes to be able to feed on different types of nutrients. This decreases competition between different species and increases the chances of their survival. The core of the Grackle algorithm is based on this successful evolutionary strategy.

While the method proposed in this paper is generic, we focus our evaluation on SMT solvers, in particular, on a recent SMT solver Bitwuzla [24]. There are ample connections between SMT and other types of automated reasoning [1,8]. For example, the *bitvector theory* [5] enables reasoning about exact representation of numbers in a computer, i.e., arithmetic modulo 2^n and exact modeling of floating-point numbers. Bitwuzla is the winner of the quantifier-free bitvector (QFBV) category of the SMT competition in 2021.[1] Hence Bitwuzla is a reasonable choice for the experiments. Moreover, we can expect the results with other solvers to be similar since none of the proposed methods are specific to Bitwuzla.

Related Work. Automatic portfolio selection and parameter tuning has long history in automated reasoning. Within the SAT community, the use of machine learning (ML) methods for the selection of a solver from a portfolio was popularized by Leyton-Brown et al. and their system called SATzilla [32]. In the context ATPs, various ML methods for strategy invention were developed [16,31,19,18].

Our work is mostly related to various approaches using optimization and ML methods for solver selection and scheduling, especially in the domain of SMT. Scott et al. developed a system called MachSMT [29] which selects a solver from a portfolio of existing solvers based on the feature representation of the given problem. Similarly to us, they use *Bag of Words* features and reduce the

[1] https://smt-comp.github.io/2021/

dimensionality with PCA. Balunovic et al. [2] use imitation learning techniques to schedule strategies within the Z3 SMT solver; the approach targets a domain specific strategy language of Z3 and therefore strategies are not understood in the same sense as in this paper. Similarly, Ramírez et al. [26] use an evolutionary algorithm to generate strategies for the Z3 solver. For an overview of related use cases of ML methods see one of the survey papers [7,21,30].

Satisfiability Modulo Theories (SMT). SMT solvers are the driving force behind software verification, testing, or synthesis, among others [6,27,13,14]. These applications often require repeated queries to an SMT solver. This means that quick response times of the solver are paramount. An SMT solver receives as input a formula and responds if it is satisfiable or not. Since the problem is generally undecidable, solvers often timeout or give up.

The language and the semantics of the given formula depends on the theories being used. For instance, the formula $(3 < x) \wedge (x < 4)$ is satisfiable in the theory of real arithmetic but unsatisfiable in the theory of integer arithmetic. The language and possible theories are standardized in the *SMT-LIB standard* [5]. A repository of benchmarks is maintained in the *SMT-LIB* [4]. A combination of theories is called a *logic*. For instance, the logic `UFNIA` supports uninterpreted functions (theory `UF`) and nonlinear integer arithmetic (theory `NIA`). Hence, it is mandatory for each problem file to specify the intended logic in the header.

Contributions. The system Grackle is presented for the first time in this work. While BliStr [31], the ancestor of Grackle, is a strategy invention system for the ATP prover E [28], Grackle supports an arbitrary solver. Moreover, Grackle adds support for an additional external tuner and implements additional features like alternative strategy selection modes and a non-atavistic behavior. Advancements of Grackle over BliStr are detailed in Section 3 and they are experimentally evaluated in Section 5. The next contribution is a strategy selector designed to select the best problem-specific strategy from the portfolio of strategies invented by Grackle. The selector is described in Section 4 and evaluated in Section 6.

2 Grackle: Strategy Portfolio Invention System

Grackle[2] is a generalization of the system called BliStr [31], which is based on the same evolutionary algorithm motivated in Section 1. BliStr is a strategy portfolio invention system for automated theorem prover E [28]. Its first successor, BliStrTune [19], is an extension of BliStr to handle larger strategy space by a hierarchical strategy invention. The second successor, EmpireTune [18], is an extension that additionally handles another ATP called Vampire [22]. Therefore, Grackle is the third successor of BliStr, and apart of the generalization from E/Vampire to a generic solver, it implements other interesting features. This section describes the core of the evolutionary algorithm common both to BliStr and Grackle. Section 3 describes novel features that are specific to Grackle only.

[2] https://github.com/ai4reason/grackle

Algorithm 1: GrackleLoop($\mathcal{S}, \mathcal{P}, \beta$)

```
$\Phi_{strats} \leftarrow \mathcal{S}$        // Initialize the state $\Phi$, and set the initial strategies
loop
    Evaluate($\mathcal{P}, \Phi, \beta$)        // Evaluate all strategies on $\mathcal{P}$ (1)
    $\Phi_{cur} \leftarrow$ Reduce($\mathcal{P}, \Phi, \beta$)        // Select the current generation of strategies (2)
    $s \leftarrow$ Select($\mathcal{P}$, $\Phi$,$\beta$)        // Select the strategy to improve (3)
    if $s$ is None then return $\Phi$        // Termination criterion
    $s_0 \leftarrow$ Specialize($s, \mathcal{P}, \Phi, \beta$)        // Improve $s$ on its best problems (4)
    $\Phi_{strats} \leftarrow \Phi_{strats} \cup \{s_0\}$        // Extend the set of strategies
```

Fig. 1. An outline of the Grackle strategy portfolio invention loop.

Figure 1 outlines the core BliStr/Grackle strategy portfolio invention loop. The input of the algorithm is a non-empty initial set of strategies $\mathcal{S}$, and the set of target problems $\mathcal{P}$. The argument β collects additional hyperparameters detailed below. The variable Φ encapsulates the current state, including, for example, the set of all strategies invented so far (Φ_{strats}). The state Φ might be modified during function calls inside the loop, while all the other variables are immutable (read-only). The current state Φ is also the output of the algorithm.

The loop consists of four basic steps. At first, all the known strategies are evaluated on all problems $\mathcal{P}$, and the results are stored in the state Φ. In the second step, a subset of strategies is selected as a current generation (Φ_{cur}), and, in the third step, one of the strategies (s) is selected for specialization. In the fourth step, the selected strategy is specialized for a subset of problems using an external tool for parameter tuning and algorithm configuration. We do not allow the same strategy to be specialized on the same problems more than once. Since we consider only finite sets of problems and possible strategies, the loop must eventually terminate because sooner or later we will run out of strategies to specialize. In practice, however, we allow the user to set the runtime limit by the hyperparameter $\beta_{timeout}$. Detailed description of the individual steps follows.

Step 1: Generation Evaluation (Evaluate)**.** In the first phase, all strategies (Φ_{strats}) are evaluated on all target problems $\mathcal{P}$. This involves running the solver on each problem with a time limit β_{eval} yielding (1) the overall result (solved/unsolved) and (2) a number representing the length of the run (runtime). Both the time limit and the runtime can be specified as a CPU time or as some abstract time (such as the number of instructions) if that is supported by the solver. This information is stored in the state Φ to avoid duplicate evaluations.

Step 2: Generation Reduction (Reduce)**.** In the next step, we select the current generation of strategies Φ_{cur} from all strategies Φ_{strats}. First, we compute for each strategy s its set of best problems $\mathcal{P}_s$, that is, the set of all problems from $\mathcal{P}$ on which s is the best strategy. In the case the best strategy of problem p is not unique, we randomly select one and mark it as the best. Then we select strategies

$s \in \Phi_{strats}$ with at least β_{bests} best-performing problems, that is, with $|\mathcal{P}_s| \geq \beta_{bests}$. From the selected strategies, we take only β_{tops} best strategies, where the strategies are compared by the number of best-performing problems ($|\mathcal{P}_s|$). The first restriction keeps only well-performing strategies (strong individuals) and removes redundant strategies (since every solved problem is exactly in one $\mathcal{P}_s$). The second restriction reduces the count of strategies, keeping the size of Φ_{cur} within the selected bound ($|\Phi_{cur}| \leq \beta_{tops}$). This prevents invention of a large number of over-specialized strategies. The function Reduce is depicted in Figure 4 and further discussed in Section 3.

Step 3: Strategy Selection (Select)**.** The next step is to select a single strategy for specialization. As a rule, no strategy can be specialized on the same problems more than once within one execution of the GrackleLoop algorithm. Because the sets of best-performing problems vary in time, the same strategy can be improved more than once, but only on different problems. Our default selection approach is to prefer specialization on diverse problems. Therefore, we prefer to improve strategies whose best-performing problems have not been used for specialization very often. In more detail, for each problem $p \in \mathcal{P}$, we keep a problem specialization counter $\Phi_{spec,p}$ that is increased by $1/|\mathcal{P}_s|$ whenever a strategy s is specialized on p (that is, when $p \in \mathcal{P}_s$). We select the strategy s with (currently) the lowest average $\Phi_{spec,p}$ over its best-performing problems $\mathcal{P}_s$. Ties are broken by higher $|\mathcal{P}_s|$. If no strategy can be selected, the algorithm terminates. The function Select is depicted in Figure 3 in Section 3.

Step 4: Strategy Specialization (Specialize)**.** In this phase, Grackle invents a new strategy by specializing on a subset of problems. The strategy specialization is done by an external parameter tuning software. BliStr uses the ParamILS [17] automated algorithm configuration framework, while Grackle additionally supports the SMAC3 [23] framework. The strategy s is always specialized on its best-performing problems $\mathcal{P}_s \subseteq \mathcal{P}$. Given a strategy s and its set of best-performing problems $\mathcal{P}_s$, the external tuner is launched to find a strategy s_0 with an improved performance on $\mathcal{P}_s$. The idea behind this is that s_0 will become even better than s on $\mathcal{P}_s$, and this shall allow additional problems outside of $\mathcal{P}_s$ to be solved. The external tuner is always launched for a specific wall-clock time limit $\beta_{improve}$. The function Select is depicted in Figure 2 in Section 3.

3 Making the Grackle Fly

This section describes what needs to be done to use the Grackle system and the main differences between Grackle and other members of the BliStr family. Apart from generalization to an arbitrary solver, main Grackle features introduced in this work are alternative methods of strategy selection and generation reduction.

Solver Wrapper. An improvement of Grackle over BliStr is that Grackle is a generalization to an arbitrary solver. To use Grackle with a selected solver, a simple wrapper function must be implemented in Python. This function takes a single problem filename together with a strategy as arguments and launches the specified solver strategy on the specified problem. Then, it must process the solver output and return the result status (solved/unsolved) and performance measurement. The performance measurement can be an arbitrary number in selected units, for example, the CPU time in seconds, or any reasonable abstract performance metric.

Grackle currently implements a solver wrapper for ATP provers E [28], Vampire [22], and Lash [10], and for SMT solvers CVC5 [3] and Bitwuzla [24]. In this paper, we focus on Bitwuzla which is used for evaluation in Section 5 and Section 6.

Parametrization of the Strategy Space. Apart from the solver wrapper, the space of all considered solver strategies must be described. This strategy space parameterization is passed to one of the supported external parameter tuners that are used to specialize a strategy to specific problems. As noted above, Grackle supports ParamILS [17] and SMAC3 [23] frameworks as external tuners. ParamILS employs an *iterated local search* (ILS) from the initial strategy, occasionally perturbing a strategy to escape from a local optimum. SMAC3 is based on Bayesian optimization in combination with an aggressive racing mechanism to efficiently decide which of the two given strategies performs better. Both frameworks are capable of finding a strategy that performs well on specific target problems.

Furthermore, both ParamILS and SMAC3 use a similar mechanism to describe the space of strategies. Since the mechanism used by SMAC3 is a subset of the one used by ParamILS, we simply use the ParamILS style to accommodate both tuners. The strategy space is described by a finite set of parameters, where each parameter is assigned a finite domain of possible values and the default value. The strategy space can be additionally pruned by specification of *conditional arguments* and *forbidden values*. Conditional arguments specify dependencies among arguments, and forbidden values allow us to specify combinations of parameter values which are banned in a single strategy. Both frameworks additionally require the user to provide a solver wrapper and a performance metric. These are, however, automatically derived from Grackle's solver wrapper. During the specialization of a strategy, the external tuner launches the solver with various strategies. Grackle's hyperparameter β_{cutoff} controls their runtime.

Parallel Tuner Execution. Both ParamILS and SMAC3 provide partial support for parallel execution to speed up the tuning. In the case of ParamILS, multiple independent instances are simply launched in parallel and, thanks to the randomized nature of the algorithm, each instance traverses the space of strategies in a different order. Each instance reports its progress as a triple (s_0, q_0, n_0), where s_0 is the best strategy found so far, and q_0 is the quality of s_0

based on evaluation on n_0 problems. The number n_0 only increases over time as new results are acquired. Finally, we select the strategy with the best quality q_0 among the different parallel runs.

SMAC3 provides similar, but improved, support for parallelization. Again, multiple instances are launched in parallel, but the instances share a common database of results and thus avoid duplicate solver evaluations.

The external tuner, called to specialize strategy s, is always launched with the wall-clock time limit specified by Grackle's hyperparameter $\beta_{improve}$. This time limit is fixed and does not reflect the size of the set of problems used for the specialization ($\mathcal{P}_s$). It can be expected that with smaller problem sets, the tuner can be launched with smaller time limits and still achieve equivalent results. Therefore, Grackle additionally implements the ParamILS extension with restarts and automated termination. We call this extension ResParamILS.

In ResParamILS, multiple ParamILS instances are launched in parallel with the same initial strategy s and the set of target problems $\mathcal{P}_s$. ResParamILS keeps checking the progress of individual instances and waits for the first ParamILS instance to report a strategy s' evaluated on all problems $\mathcal{P}_s$. That is, it waits for some instance to report a triple (s', q', n') where $n' = |\mathcal{P}_s|$. Then ResParamILS enters a stabilization phase and waits for t seconds, where t is the wall-clock time elapsed so far. This stabilization phase allows other instances to evaluate the best strategy on all problems $\mathcal{P}_s$. Then, only the best ParamILS instance is kept running while the other instances are terminated. The terminated instances are then restarted with the best strategy s' found so far as the initial configuration, but keeping the initial set of problems $\mathcal{P}_s$. This process is then iterated and ends when the quality of the best strategy stops improving, that is, when no better strategy can be found. In this way, ResParamILS tries to detect a plateaued state. The hyperparameter $\beta_{improve}$ can be still used as an overall tuning limit.

BliStr does not support parallel execution of the tuner. It was first added in BliStrTune but without ResParamILS. ResParamILS was already partially implemented in EmpireTune, but without any evaluation. Grackle is the first system of the BliStr family to support SMAC3. The number of parallel runs in Grackle is controlled by the hyperparameter β_{cores}.

Strategy Selection Mechanisms. As described in Section 2, we prefer specialization to problems that have not been used very often for specialization. This is implemented by keeping a global problem specialization counter $\Phi_{spec,p}$ for every problem p. This counter is initialized to zeros and is updated with every call to the function Specialize, as described in the Figure 2.

When selecting the strategy to be specialized, the strategies are compared by averaging problem specialization counters over the best-performing problems $\mathcal{P}_s$. For each strategy s, we compute the average problem specialization index $\mathcal{C}_s$ as described in Figure 3. Since we prefer problems that are not often used for specialization, we prefer smaller values of $\mathcal{C}_s$. Grackle provides several ways to use this index to select the strategy to specialize.

```
Function Specialize($s, \mathcal{P}, \Phi, \beta$)
  $\mathcal{P}_s \leftarrow \{p \in \mathcal{P} \mid s$ is the best strategy on problem $p$ among strategies $\Phi_{cur}\}$
  $s_0 \leftarrow$ ExternalTuner($s, \mathcal{P}_s, \beta_{cutoff}$)          // launch the external tuner
  for $p \in \mathcal{P}_s$ do          // update the problem specialization counters $\Phi_{spec}$
      $\Phi_{spec,p} \leftarrow \Phi_{spec,p} + (1/|\mathcal{P}_s|)$
  return $s_0$
```

Fig. 2. Specialize the strategy s on its best problems $\mathcal{P}_s \subseteq \mathcal{P}$.

```
Function Select($\mathcal{P}, \Phi, \beta$)
  for $s \in \Phi_{cur}$ do
      $\mathcal{P}_s \leftarrow \{p \in \mathcal{P} \mid s$ is the best strategy on problem $p$ among strategies $\Phi_{cur}\}$
      $\mathcal{C}_s \leftarrow (\sum_{p \in \mathcal{P}_s} \Phi_{spec,p})/|\mathcal{P}_s|$          // average problem specialization counters
      $\mathcal{Q}_s \leftarrow (\mathcal{C}_s, -|\mathcal{P}_s|)$          // quality pairs are compared lexicographically
  $\mathcal{S} \leftarrow \{s \in \Phi_{cur} \mid s$ has not been specialized on $\mathcal{P}_s$ yet$\}$          // skip already done
  return $\arg\min_{s \in \mathcal{S}} \mathcal{Q}_s$
```

Fig. 3. Select the strategy to be specialized.

Default mode. The *default selection mode* is the selection mechanism from BliStr, where the strategies with lower $\mathcal{C}_s$ are preferred. In the case of equal values, which happens always in the first iteration as the counters are zeroed, the strategies are compared by their performance, that is, by $|\mathcal{P}_s|$. In the algorithm, this is implemented by constructing the quality pair $\mathcal{Q}_s = (\mathcal{C}_s, -|\mathcal{P}_s|)$. The quality pairs are then compared lexicographically, and the strategy s with the lowest $\mathcal{Q}_s$ is selected for specialization. Since the size of $\mathcal{P}_s$ is reversed in $\mathcal{Q}_s$, we prefer stronger strategies in the case of equal values of $\mathcal{C}_s$. In particular, in the first iteration, the strongest strategy s will be specialized. The default mode is the only strategy selection implemented in BliStr.

Reverse mode. In practice we often observe that strong strategies are born out of weak ones. Therefore, Grackle additionally provides a way to begin the specification with the weakest strategy. In the *reverse selection mode*, this is implemented by using '$\arg\max$' instead of '$\arg\min$' in the last line of function Select.

Weak mode. The reverse mode, however, prefers specialization on problems already used for specialization, because it prefers higher values of $\mathcal{C}_s$ in the first element of a quality pair $\mathcal{Q}_s$. Therefore, we implement the *weak selection mode* which uses '$\arg\min$' as in the default mode, but it constructs the quality pair $\mathcal{Q}_s$ as $\mathcal{Q}_s = (\mathcal{C}_s, |\mathcal{P}_s|)$. In this way, we prefer weak strategies, but we still prefer problems not often used for specialization.

Random mode. Grackle additionally implements a *random selection mode* where the strategy is selected randomly.

```
Function Reduce($\mathcal{P}, \Phi, \beta$)
  $\mathcal{G} \leftarrow \Phi_{strats}$                                  // start with all strategies known so far
  for $s \in \mathcal{G}$ do                        // compute the best problems for every strategy
      $\mathcal{P}_s \leftarrow \{p \in \mathcal{P} \mid s$ is the best strategy on problem $p$ among strategies $\mathcal{G}\}$
  $\mathcal{G} \leftarrow \{s \in \mathcal{G} \mid$ if $|\mathcal{P}_s| \geq \beta_{best}\}$                          // keep only strong individuals
  $\mathcal{G} \leftarrow [s \in \mathcal{G} \mid$ sort $\mathcal{G}$ by decreasing $|\mathcal{P}_s|]$  // list strategies sorted by performance
  $\mathcal{G} \leftarrow \{s \mid s$ is among the first $\beta_{tops}$ strategies in $\mathcal{G}\}$  // keep only best strategies
  return $\mathcal{G}$
```

Fig. 4. Selection of the current generation of strategies.

Evaluation. All Grackle ancestors support only the default selection mode. The various Grackle selection methods are experimentally evaluated in Section 5.

Atavistic and Non-Atavistic Modes. In genetics, *atavism* is a recurrence of a trait typical for ancestors but not apparent in the current generation. In our context, it can happen that a strategy disappears from the current generation Φ_{cur} but suddenly reappears in one of the following iterations. This happens due to the selection of the current generation in the function Reduce depicted in Figure 4.

In BliStr and other Grackle ancestors, the current generation is selected out of all known strategies Φ_{strats}. Strategies are filtered by their performance $|\mathcal{P}_s|$ using hyperparameters β_{best} and β_{tops}. Note that $\mathcal{P}_s$ in Reduce is computed with respect to all strategies Φ_{strats}, while in functions Select and Specialize it is computed with respect to the current generation Φ_{cur} only. Since the count $|\mathcal{P}_s|$ can only decrease during the execution of the Grackle loop, a strategy once rejected due to the limit β_{best} can never reappear in future generations. However, a strategy rejected due to the limit β_{tops} can appear in a future generation because the order of the strategy can change. This behavior, called *atavistic*, is the behavior implemented in BliStr.

Grackle additionally supports a *non-atavistic* behavior, where the next generation is selected out of the strategies of the previous generation. This is implemented by setting $\mathcal{G}$ to Φ_{cur} instead of Φ_{strats} in the first line of Reduce. Furthermore, Φ_{cur} is initialized with the initial strategies, and any specialized strategy (any output of Specialize) is always added to the current generation.

4 Strategy Selection with Boosted Decision Trees

Given a large portfolio of strategies obtained by running Grackle, we want to select the best strategy for every problem within the given benchmark set. We achieve this by training a ranking model to rank strategies using features extracted from the problem in question. Then, for an arbitrary input problem, we run the strategy that has the highest rank according to the trained model.

Feature Extraction. We use a simple *Bag of Words* (BOW) as a feature representation of each problem/formula. To compute this representation, we parse the given formula and compute the counts of every unique word. The set of possible words consists of logical operators of the SMT language together with keywords of the given logic. We ignore the concrete names of variables and functions and the concrete values of constants. For example, we do not count how many times a concrete value (such as 1) appears within the formula. Instead, we count the number of occurrences of any constant of a given type (i.e., an integer numeral).

Given a fixed order of the possible words which could appear in an SMT formula, the BOW representation of the formula is a vector in which the n-th element represents how many times the n-th possible word appears within the formula. If we combine multiple logics together, then for a concrete problem (belonging to a specific logic), most of the possible words will not be used and the feature vector will be sparse. Therefore, we reduce the dimension of these vectors by *Principal component analysis* (PCA), and for any new problem, we project its original feature vector to the obtained principal vectors. The simplicity of the feature extraction phase results in a negligible computation time, which is crucial in the setting we are interested in.

Strategy Ranking. To select the strategy on a per-instance basis, we train a ranking model that takes the extracted feature vector and a categorical variable whose possible values correspond to different strategies as input. The model outputs the rank of the strategy for the given problem. For every new problem, we select the strategy with the highest rank. As a ranking model, we use LightGBM [20] with a default setting and the LambdaRank [11] objective function and train it for 1000 iterations[3].

We follow two different procedures when creating labels for the training dataset, depending on the timeout parameter. When the timeout is small (1 s in our case), we set the rank of a given strategy to 1 or 0 depending on whether the strategy solves the given problem before the timeout or not, respectively. When the timeout is set to higher values (10 s or 60 s in our case), we divide the timeout into five intervals and set the rank of the strategy according to the interval in which it solves the problem. Concretely, if the strategy does not solve the problem before the timeout, we set its rank to 0. If it solves the problem during the last interval (whose endpoint corresponds to the timeout), we set its rank to 1 and so on, until the first interval, which corresponds to a rank equal to 5. In simple words, a shorter solving time corresponds to a higher rank.

For the case of longer timeout (10 s or 60 s), it holds that for any given strategy, the solving times are not distributed uniformly, as can be seen in the lower right part of Figure 5 that contains a histogram of the solving times under 60 seconds. The histogram shows that the solving times follow an exponentially decreasing trend. To counteract this nonuniform distribution, we set the endpoints

[3] We also tried to use a Graph Neural Network which processed formulas represented as directed acyclic graphs but LightGBM ranking with BOW representation provided the best tradeoff in terms of accuracy and speed.

of the intervals according to a power function. That is, we set the endpoint of an interval n to n^p for some $p \in \mathbb{R}$. If we want to have N intervals in total, p is chosen so that $N^p = timeout$. For example, $N = 5$ and $timeout = 60$ yield $p = \log_5(60) \simeq 2.54$ and interval endpoints: $(1, 5.8, 16.3, 34, 60)$.

The rationale behind the different treatment of the labels for the case of the longer timeout (10 and 60 seconds) is that it may be easier for the ranking model to distinguish whether the strategy solves the problem in 1 second or 30 seconds compared to whether it solves it in 200 milliseconds or 600 milliseconds. In our experiments, having multiple ranks for a longer timeout led to better results.

Another option to obtain the ranks of the strategies would be to sort the strategies by solving time and set the rank of each strategy to its order. This would be harder to learn because the prediction would need to be more precise.

5 Evaluation of Grackle Strategy Invention

In this section we experimentally evaluate Grackle strategy invention on an SMT solver Bitwuzla, and on a benchmark of 3000 problems used in the SMT competition in 2021. Bitwuzla's configuration used in the SMT competition is called *smt-comp-mode* and it serves as a baseline in our experiments.

We select the benchmark problems as follows. In the competition, Bitwuzla was launched with 13605 problems and solved 13291 while only 342 problems were not solved. The time limit per problem in the SMT competition was 20 minutes. First, we include all 342 unsolved problems in our benchmark. Next, we remove all problems with runtime smaller than 100 ms to filter out trivial problems. From the remaining problems, we randomly select 2658 problems to obtain the set of 3000 benchmark problems. This set is divided into 2000 training and 1000 testing problems. All experiments in this section are performed with the 2000 training problems, while the testing problems are used for the final evaluation in Section 6.

To use Bitwuzla with Grackle we need to parametrize the strategy space. We manually select 39 parameters and their domains based on our intuition. We have 28 boolean parameters, and the overall size of the strategy space is about 10^{15}. To test the parametrization, we launch several instances of SMAC3 without using Grackle. This gives us 20 Bitwuzla strategies that we use as the initial strategies for all Grackle runs described below. During tuning, Bitwuzla is always launched with a 1 second time limit per problem. The best of the initial strategies solves 964 (out of 2000) training problems with 1 second time limit, while Bitwuzla's *smt-comp-mode* solves 906. The union of problems solved by all initial strategies is 1345.

The first experiment tests different external tuners supported by Grackle, that is, SMAC3, ParamILS, and ResParamILS (see Section 3). We launch one Grackle instance for each tuner with the runtime limit of 24 hours and with 8 cores per instance. The tuning time for one specialization is set to 5 minutes ($\beta_{improve}$). Individual Bitwuzla runs are limited to 1 second, both during evaluations (β_{eval}) and specializations (β_{cutoff}). We use the non-atavistic behavior,

tuner	*solved*		*specializations*		*runtime* [*hh*:*mm*]			*termination*
	total	*new*	*success*	*all*	*eval.*	*spec.*	*total*	
SMAC3	1449	104	28	128	01:29	10:45	12:15	regular
ParamILS	1459	114	161	186	08:07	15:35	23:56	timeout
ResParamILS	1452	107	194	211	09:58	13:39	23:56	timeout

Table 1. Evaluation of external tuners supported by Grackle.

and we restrict the size of the current generation to 30 (β_{tops}). We require every strategy to outperform other strategies in least at 3 problems (β_{bests}).

The results are summarized in Table 1. The column *total* describes the number of problems solved by all invented strategies, and the column *new* shows how many of them are not solved by the initial strategies. The column *success* presents the number of strategies invented by specialization, that is, the count of successful specializations. It can happen, that the outcome of a specialization is some strategy that is already known, that is, Specialize$(s, \mathcal{P}, \Phi, \beta) \in \Phi_{strats}$, in which case we consider the specialization as a failure. The column *all* shows the total number of specializations performed, including failed ones. The columns *runtime* describe Grackle run times in hours and minutes. First two columns describe the time used for evaluations (*eval.*) and the time used for specializations (*spec.*). The *total* runtime additionally covers time used for the reductions and selections of the strategies. The last column describes whether Grackle terminates because no strategy can be specialized or because of the timeout.

First, we observe that the numbers of solved problems are quite similar. We can see that SMAC3 specializations failed much more often. It seems that SMAC3 tends to return the input strategy unless it finds a strictly better one. On the other hand, this happens rarely with ParamILS, where the local search deviates from the initial input strategy quite early. Due to that, ParamILS and ResParamILS managed to invent more strategies, which is a behavior favored by Grackle. Furthermore, the results of ParamILS and ResParamILS can be improved by extending the time limit. We also see that ResParamILS was able to perform more iterations with quite a smaller specialization runtime. This is thanks to its automated termination feature.

While SMAC3 invents 28 strategies, only 20 are needed to cover all solved problems. For ParamILS, this is 21 and for ResParamILS it is 20. From this we can conclude that the invented strategies have a similar strength. Recall that the initial strategies solve 1345 problems. The total number of problems solved by all three runs together is 1481. This means that different tuners invent complementary strategies and that there is no clear winner among the tuners.

In the second experiment, we test different strategy selection and reduction modes from Section 3. Here, we restrict our attention to ResParamILS. We test *atavistic* and *non-atavistic* behaviors, combined with three modes of strategy selection (*default*, *reverse*, *weak*). This gives us six different Grackle runs. Otherwise, we use the same Grackle hyperparameters as in the first experiment.

atavistic	*selection*	*solved* *total*	*new*	*specializations* *success*	*all*	*runtime* [*hh*:*mm*] *eval.*	*spec.*	*total*	*termination*
no	default	1452	107	194	211	09:58	13:39	23:56	timeout
no	reverse	1472	127	220	237	12:11	11:22	23:58	timeout
no	weak	1429	84	160	174	09:28	14:17	24:00	timeout
yes	default	1463	118	199	208	09:57	13:42	24:00	timeout
yes	reverse	1425	80	196	212	11:27	12:08	23:55	timeout
yes	weak	1421	76	169	195	09:11	14:32	23:58	timeout

Table 2. Evaluation of selected strategy selection modes.

The results are presented in Table 2. The first row is the same as the ResParamILS row from the first experiment. We can see that the reverse selection with the non-atavistic behavior solves most problems and invents most strategies. It performs quite differently with the atavistic behavior, where it solves fewer problems even though it invents a large number of strategies. This suggests that the atavistic mode favors the invention of redundant strategies. The weak selection solves the least problems, but it spends the most time by specializations. The total number of problems solved by all six runs is 1500. This again suggests that there is a certain complementarity among the methods.

6 Evaluation of Strategy Selection

In the last Section 5 we have described 9 different Grackle runs. We additionally perform several Grackle runs with different hyperparameters, for example, increasing the Bitwuzla time limit to 5 seconds (β_{eval} and β_{cutoff}). Together, we collect more than 5000 different Bitwuzla strategies, and by iterative greedy cover construction, we select 140 complementary strategies. We evaluate these strategies on the training problems with a 60 second time limit per problem and strategy. This gives us training data for a strategy selector that attempts to select the best strategy for a specific input problem.

While it often makes sense to alternate several strategies within a given time limit, in our context of quantifier-free bit vectors, it is often best to select a single strategy and run it exclusively until the time limit is reached. Therefore, we attempt to construct the selector of a single strategy. We focus on smaller time limits (below 1 minute), because we develop methods for a machine-human interaction, keeping in mind limited time resources and impatience of human users. In particular, we test the strategy selection with 3 different time limits (1, 10 and 60 seconds). The selectors are evaluated on the 1000 testing problems.

To evaluate the selector, we count the number of solved problems for a given timeout and compute the *PAR-2* score (Penalized Average Runtime) commonly used to compare different solvers. The score is the sum of the runtimes of solved problems plus twice the timeout for each unsolved problem. The qualitative

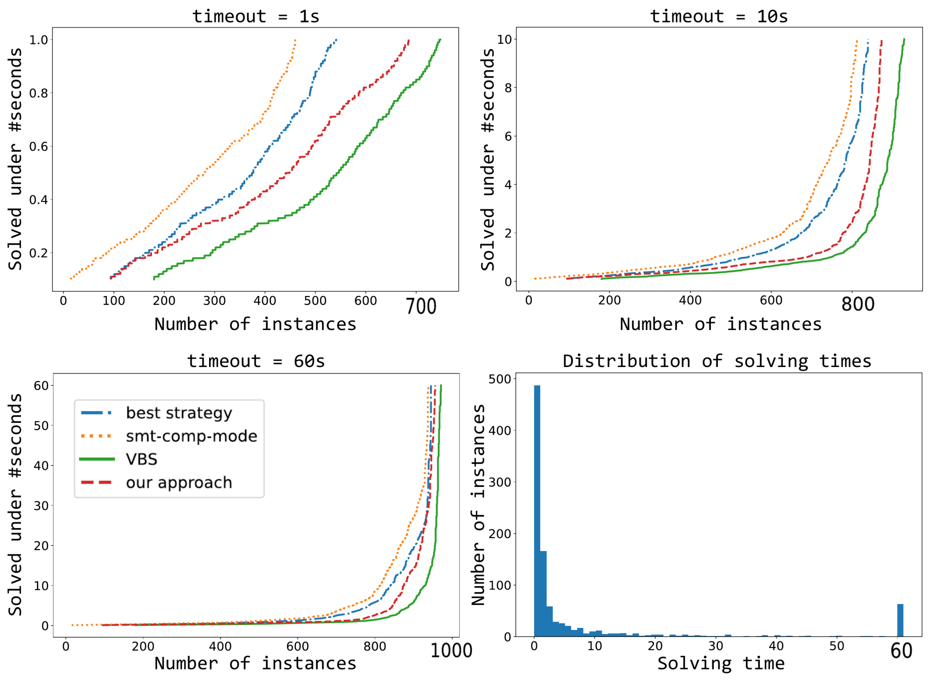

Fig. 5. *Top left, top right, bottom left*: Cactus plots for different timeouts (1s, 10s, and 60s, respectively). The plot shows a comparison of our approach against the virtual best strategy (VBS), the baseline *smt-comp-mode* strategy, and the best invented strategy from the portfolio. *Bottom right*: Histogram of solving times of one selected strategy. The bar at 60 seconds corresponds to unsolved problems.

and quantitative results can be seen in Figure 5 and Table 3, respectively. We compare the strategy selector with the default *smt-comp-mode* strategy, the best strategy from the invented strategies, and the virtual best strategy. The virtual best strategy is a hypothetical selector which would always pick the best performing strategy. The results show that the gain is largest when the time limit is small and there is a substantial gap between the best performing strategy and the virtual best strategy. We observe the clear superiority of our selector.

7 Conclusions and Future Work

We have presented a method for the automated configuration of automated reasoning solvers for specific target problems. We have evaluated the method on the SMT solver called Bitwuzla by targeting the solver to a subset of SMT-LIB benchmarks. We have invented a large amount of Bitwuzla strategies using the Grackle system, that is described for the first time in this work. We have achieved a substantial improvement over the default Bitwuzla mode. The strength of our system is that it both invents new strategies as well as selects the best one to

time limit	*solved by*	*best strategy*		*our selector*		*solved by*
	smt-comp-mode	*solved*	*PAR2+*	*solved*	*PAR2+*	*VBS*
1s	460	541	13.59%	689	39.92%	746
10s	813	840	16.38%	873	50.50%	929
60s	938	945	19.04%	956	46.76%	971

Table 3. Results of the evaluation. VBS stands for virtual best solver. *PAR2+* is a relative improvement of the PAR-2 score to the score of *smt-comp-mode* in percent.

be used on a given problem. To the best of our knowledge, this is the first time such approach was applied to an SMT solver.

For future work, we would like to inspect alternative methods of strategy selection and generation reduction in Grackle. For example, Grackle also supports the specialization of a strategy s on unsolved problems by interleaving the best problems $\mathcal{P}_s$ with similar but unsolved problems. This feature has not yet been tested. Additionally, we would like to test our methods on other solvers and to construct strategy schedules that alternate execution of multiple strategies.

References

1. Ábrahám, E., Abbott, J., Becker, B., Bigatti, A.M., Brain, M., Buchberger, B., Cimatti, A., Davenport, J.H., England, M., Fontaine, P., Forrest, S., Griggio, A., Kroening, D., Seiler, W.M., Sturm, T.: SC^2: Satisfiability checking meets symbolic computation - (project paper). In: CICM (2016)
2. Balunovic, M., Bielik, P., Vechev, M.T.: Learning to solve SMT formulas. In: NeurIPS. pp. 10338–10349 (2018)
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: TACAS (1). vol. 13243. Springer (2022)
4. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., The University of Iowa (2017), available at `www.SMT-LIB.org`
6. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, pp. 825–885. IOS Press (2009)
7. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. European J. of Operational Research (2020)
8. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. J. Autom. Reason. **51**(1) (2013)
9. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formalized Reasoning **9**(1), 101–148 (2016)
10. Brown, C.E., Kaliszyk, C.: Lash 1.0 (system description). CoRR **abs/2205.06640** (2022)

11. Burges, C.J.C., Ragno, R., Le, Q.V.: Learning to rank with nonsmooth cost functions. In: NIPS. pp. 193–200. MIT Press (2006)
12. The Coq Proof Assistant, http://coq.inria.fr
13. de Moura, L., Bjørner, N.: Applications and challenges in satisfiability modulo theories. In: Workshop on Invariant Generation (WING) (2012)
14. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)
15. Grabowski, A., Korniłowicz, A., Naumowicz, A.: Mizar in a nutshell. J. Formalized Reasoning **3**(2), 153–245 (2010)
16. Holden, E.K., Korovin, K.: Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In: CICM. vol. 12833. Springer (2021)
17. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artificial Intelligence Research **36** (2009)
18. Jakubův, J., Suda, M., Urban, J.: Automated invention of strategies and term orderings for Vampire. In: GCAI (2017)
19. Jakubův, J., Urban, J.: BliStrTune: hierarchical invention of theorem proving strategies. In: CPP (2017), http://doi.acm.org/10.1145/3018610.3018619
20. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.: LightGBM: A highly efficient gradient boosting decision tree. In: NIPS (2017)
21. Kerschke, P., Hoos, H.H., Neumann, F., Trautmann, H.: Automated algorithm selection: Survey and perspectives. Evolutionary computation **27**(1), 3–45 (2019)
22. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: CAV. LNCS, vol. 8044, pp. 1–35. Springer (2013)
23. Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., Hutter, F.: SMAC3: A versatile bayesian optimization package for hyperparameter optimization (2021)
24. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), https://arxiv.org/abs/2006.01621
25. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer (2014)
26. Ramírez, N.G., Hamadi, Y., Monfroy, É., Saubion, F.: Evolving SMT strategies. In: ICTAI (2016)
27. Reynolds, A., Kuncak, V., Tinelli, C., Barrett, C.W., Deters, M.: Refutation-based synthesis in SMT. Formal Methods Syst. Des. **55**(2) (2019)
28. Schulz, S.: System description: E 1.8. In: LPAR. pp. 735–743 (2013)
29. Scott, J., Niemetz, A., Preiner, M., Nejati, S., Ganesh, V.: MachSMT: a machine learning-based algorithm selector for SMT solvers. In: TACAS (2020)
30. Talbi, E.G.: Machine learning into metaheuristics: A survey and taxonomy of data-driven metaheuristics. ACM Computing Surveys (2020)
31. Urban, J.: BliStr: The Blind Strategymaker. In: Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015 (2015)
32. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32** (2008)