Adaptive Algorithm Selection for Btor2 Verification Tasks
Google Summer of Code 2024 Project Report

Zhengyang John Lu * john.lu2@uwaterloo.ca

Abstract

We developed the first algorithm selector, Btor2-Select, for the word-level hardware model-checking problem described in the Btor2 language. Given a Btor2 instance, Btor2-Select selects and applies the expected best verifier(s) from a pool of hardware and software tools. These decisions are based on various machine learning (ML) models, trained upon historical performance data, mapping from instance features to algorithm selections.

We proposed two embeddings for Btor2 instances: bag of keywords and bit-width aggregation. Two traditional algorithm-selection ML models, i.e., empirical hardness model and pairwise classifiers, were implemented in Btor2-Select. More importantly, we developed a novel algorithm selection and scheduling framework based on deep reinforcement learning. This framework allows for adaptive algorithm selection throughout the solving process, leveraging dynamic information such as previous attempts and elapsed time. Upon evaluation, the adaptive algorithm selector outperformed the best non-portfolio solver (SBS) by 24.4%. Moreover, it closed 6.69% more SBS-VBS gaps than the best non-adaptive algorithm-selection method, a common performance indicator for algorithm selectors.

All the codes, experiments, and data can be found at this GitLab repo.

1. Introduction

It has long been observed that for computationally hard problems, no single algorithm performs well on all instances, and different algorithms perform well on distinct classes of instances. To leverage such complementary strengths, machine learning (ML)–based algorithm selection is gaining popularity (Xu et al., 2012). For each given instance, an algorithm selector predicts and selects the optimal algorithm(s) from a set of complementary candidates, based on previously collected empirical performance data.

Our research focuses on algorithm selection for the hardware model-checking problem in the Btor2 format (Niemetz et al., 2018). Hardware model checking plays a crucial role in ensuring the correctness and reliability of critical hardware systems, as even minor errors in hardware designs can lead to catastrophic failures. Traditionally, hardware model checkers such as `ABC` (Brayton & Mishchenko, 2010) have been developed to verify the correctness of hardware designs against given specifications. What makes this problem more interesting is the recent development of Btor2C (Beyer et al., 2023), a Btor2-to-C translator, which allows software verifiers to be applied to hardware verification tasks as well.

We propose, to the best of our knowledge, the first algorithm selector for the hardware verification problem. Our tool, called Btor2-Select, leverages the strength of both

*. GSoC Mentors: Po-Chun Chien and Nian-Ze Lee (LMU Munich)
The author also thanks Arie Gurfinkel (UWaterloo), Vijay Ganesh (Georgia Tech), Cedric Richter (Paderborn University), and Chris Cameron (UBC) for guidance and help.

```
1 sort bitvec 3
2 zero 1
3 state 1
4 init 1 3 2
5 input 1
6 add 1 3 5
7 one 1
8 sub 1 6 7
9 next 1 3 8
10 ones 1
11 sort bitvec 1
12 eq 11 3 10
13 bad 12
```

(a) BTOR2 circuit

```
extern void abort(void);
extern unsigned char nondet_uchar();
void main() {
  typedef unsigned char SORT_1;
  typedef unsigned char SORT_11;
  const SORT_1 var_2 = 0b000;
  const SORT_1 var_7 = 0b001;
  const SORT_1 var_10 = 0b111;
  SORT_1 input_5;
  SORT_1 state_3 = var_2;
  for (;;) {
    input_5 = nondet_uchar();
    input_5 = input_5 & 0b111;
    SORT_11 var_12 = state_3 == var_10;
    SORT_11 bad_13 = var_12;
    if (bad_13) {
      ERROR: abort();
    }
    SORT_1 var_6 = state_3 + input_5;
    var_6 = var_6 & 0b111;
    SORT_1 var_8 = var_6 - var_7;
    var_8 = var_8 & 0b111;
    state_3 = var_8;
  }
}
```

(b) C program (simplified for demo)

Figure 1: An example BTOR2 circuit (a) and its translated C program (b) from Beyer et al. (2023)

hardware and software verifiers. More importantly, it can dynamically adjust the algorithm-selection choice during the solving process according to runtime information. This adaptive feature is empowered by reinforcement learning (RL). We implemented our adaptive algorithm selector BTOR2-SELECT in Python and conducted experimental evaluations on the tool. Experimental results showed that the adaptive algorithm selector outperformed the single best non-portfolio solver (SBS) by 24.4% in terms of the PAR-2 runtime score. Moreover, it closed 6.69% more gaps between the SBS and the virtual-best-solver than the best baseline non-adaptive algorithm-selection method.

All the codes, experiments, and data can be found at this GitLab repo.

2. Background

2.1 Btor2 and Btor2C

BTOR2 is a word-level model-checking format for sequential circuits. The modeling-checking problem decides whether a safety property holds on all possible executions of a circuit. BTOR2 is adopted by HWMCC (A. Biere et al., 2020), the international competitive event for hardware model checkers, as the format for the word-level track.

In a BTOR2 file, each line serves a specific purpose and is distinguished by keywords. For example, the keyword `sort` is used to define arbitrary bit-vector and array sorts. Registers and memories are specified by the keyword `state`, and their initialization and transitions are defined with the keywords `init` and `next`. The combinational circuits are described by various operators, such as `not`, `and`, and `add`. The keyword `bad` encodes negations of the safety property.

We provide a BTOR2 example in Figure 1a. This instance describes a circuit whose state is a bit-vector of width 3. In lines 2-4, the state is initialized to 0. In each iteration, the value of this state bit-vector will first be incremented by an external input (lines 5-6) and then decremented by 1 (lines 7-8). The circuit breaks the safety property if the state bit-vector equals `0b111` (lines 12-13).

2.2 Algorithm Selection

For computationally intractable problems, it is common for one algorithm to perform better than other algorithms on certain instances, while performing dramatically worse on some other instances. Rice (1976) first proposed the *algorithm selection* problem: given an instance from a particular problem, which algorithm(s) should be run to optimize the performance? One pioneering and successful application of algorithm selection is SatZilla (Xu et al., 2008), an algorithm-selection-based portfolio SAT solver that won multiple medals in both the 2007 and 2009 SAT Competitions (Le Berre et al., 2024a, 2024b). Over the decades, algorithm selection has also been widely used in areas such as SMT (Scott et al., 2023), constraint satisfaction problems (O'Mahony et al., 2008), and planning (Cenamor et al., 2014). While algorithm selection has been explored in the context of software verification (Leeson & Dwyer, 2024; Richter & Wehrheim, 2020; Richter et al., 2020; Tulsian et al., 2014), to the best of our knowledge, no prior work has investigated its application to hardware verification.

The objective of the algorithm selection is to select the optimal algorithm from a set of candidates for each instance from a problem set. However, it is impractical to efficiently find a guaranteed perfect solution to the algorithm selection problem. Instead, algorithm selectors are usually trained, upon historical performance data, to predict algorithms' performance given some cheaply computable features of an input instance. We refer to the hypothetically perfect or oracle selector as the virtual best solver (VBS). The VBS provides a performance upbound. We call the individual algorithm with the best performance among all candidates the single best solver (SBS). Realistic algorithm selectors are usually evaluated by the fraction of the VBS-SBS performance gap closed by the selector.

Below are introductions to two prevalent algorithm-selection models.

Empirical Hardness Model Empirical Hardness Model (EHM) (Leyton-Brown et al., 2002) is a predictor of an algorithm's runtime on a given instance. It is an ML model trained on the algorithm's past performance. EHMs are widely used for algorithm selection (Scott et al., 2023; Xu et al., 2008), by building an EHM for each component algorithm, and, at runtime, selecting the algorithm with the best-predicted performance for the given instance.

Cost-Sensitive Pairwise Classifier A cost-sensitive pairwise classifier (PWC) (Xu et al., 2012) is a classifier that predicts, given a pair of algorithms, which one would perform better. During training, samples are weighted by the performance difference between the pair. When used for algorithm selection, one PWC is trained for every pair of candidate algorithms. During inference, for a given instance, all PWCs are evaluated, and the algorithm that receives the highest votes is selected.

2.3 Algorithm Scheduling

Typically, an algorithm selector selects a solver and commits to it. By contrast, *algorithm scheduling* defines a sequence of solvers to be tried, each with an individual internal time limit.

Although scheduling cannot outperform the perfect algorithm selector VBS, constructing such an ideal selector is impractical. Moreover, it has been observed in many scenarios that if an instance is solvable by a particular algorithm, it is usually solved in a short time. Thus, algorithm scheduling helps hedge the bet, strategically distributing the given time budget to different solvers, thereby potentially increasing the likelihood of success.

2.4 Deep Reinforcement Learning

Reinforcement learning (RL) is a framework in which an agent learns to make sequential decisions within an environment modeled as a Markov Decision Process (MDP) (Sutton & Barto, 2018). In an MDP, the agent transitions through states by taking actions, with each transition yielding a reward. The goal is to develop a policy that maximizes cumulative rewards over time.

Deep reinforcement learning (DRL) extends RL by leveraging deep neural networks to approximate the policy or value functions, which is crucial for handling environments with high-dimensional state representations, suitable for our problem. Through DRL, agents effectively learn in complex environments, with popular algorithms like Deep Q-Networks (DQN) (Mnih et al., 2015) and AlphaZero (Silver et al., 2017) achieving superhuman performance in challenging video and board games.

2.5 Measurements of Contribution to a Portfolio

One of our research goals is to evaluate how each component verifier in our algorithm-selection portfolio contributes to the overall performance. Fréchette et al. (2016) proposed to evaluate an algorithm's contribution to a portfolio using the *Shapley values* (Shapley, 1953). The Shapley value, originating from cooperative game theory, is widely regarded as a fair measure of individual components' contribution to a coalition's performance. Among various ways to do such gain distribution, the Shapley value uniquely satisfies a set of desirable properties: efficiency, symmetry, dummy, and additivity.

The Shapley value computes the average marginal contribution of each player over all possible orders to join a coalition. Let us define a cooperative game as a pair (V, p), where $V = \{v_1, ..., v_n\}$ is a set of n players and $p : 2^V \to \mathbb{R}$ is a function that maps each coalition

of players $C \subseteq V$ to a real number $p(C)$, representing the gain of C. Then, the Shapley value of player $v_i \in V$ is calculated as:

$$\phi_{v_i} = \frac{1}{n!} \sum_{\pi \in \Pi^V} (p(C^\pi_{v_i} \cup \{v_i\}) - p(C^\pi_{v_i})) \tag{1}$$

where Π^V denotes the set of all permutations of V; $C^\pi_{v_i}$ denotes the coalition consisting of all predecessors of v_i in π.

3. Adaptive Algorithm Selection and Scheduling for Btor2

3.1 Instance Representation of Btor2

Since algorithm selection operates on an instance basis, a well-constructed representation of each problem instance is crucial. The representation shall be efficiently computable, suitable as input for ML models, and preserving essential information that distinguishes performance among solvers. With these goals in mind, we propose two types of BTOR2 instance representations: *Bag of Keywords* (BoKW) and *Bit-Width Aggregation* (BWA).

Bag of Keywords For each instance, BoKW counts the occurrence of each *keyword* from a predefined set of 69 keywords, such as `state`, `sort`, and `not`, to form the instance representation. For example, in the BTOR2 instance shown in Figure 1a, `state` appears once, `sort` twice, and `not` does not appear. The BoKW representation of this example would be a vector of 69 integers, with entries corresponding to the frequencies of `state`, `sort`, and `not` set to 1, 2, and 0, respectively.

Bit-Width Aggregation Most keywords (with the exception of `bad`, `constraint`, `fair`, `output`, and `justice`) return a variable of a certain bit-width. For example, the two `sort`'s in Figure 1a return a 3-bit vector and a 1-bit vector, respectively. BWA, instead of counting the occurrence, sums the bit-widths of all returned variables for each relevant keyword. For `bad`, `constraint`, `fair`, `output`, and `justice`, BWA simply counts the occurrence of these keywords. There, the BWA representation of the BTOR2 example would be a vector of 69 integers, with entries corresponding to `state`, `sort`, `not`, and `bad` set to 3, 4, 0, and 1, respectively.

3.2 Modelling Algorithm Selection and Sheduling

We model our algorithm selection and scheduling problem as a Markov Decision Process (MDP). An MDP is a mathematical framework in which an agent makes action decisions in a series of states, with each action leading to a state transition and a reward. The agent seeks to maximize rewards over time through choices of actions.

In our formulation of the problem, each MDP episode is the process of dynamically selecting verifiers to solve one particular BTOR2 instance b^i with an external time limit T. At each step, the agent, i.e., the algorithm selector, chooses a verifier v from a predefined set V, possibly with a specified internal time limit t, as actions. The execution of the selected verifier may lead to successfully solving the instance, or failing due to various reasons such as reaching the time limits. The dynamic runtime information collected during the solving process can be embedded in the states to help future action choices. The goal of the agent is

to successfully solve as many as possible instances and, in the meantime, minimize solving time.

Formal Description We provide a formal description of our problem MDP, defined by the tuple (S, A, D), where each component is detailed below.

- **States** S: Each state $s \in S$ represents the current status of solving a specific BTOR2 instance b^i. Every episode begins in a specific initial state s_0, which is sampled to reflect the start of solving b^i from an instance distribution D_b of interests. An s is a terminal state if the instance is successfully solved, or the external time limit T has been reached. In our current design, each state s comprises two main components: instance features and solving history. We described the different types of instance features we use in Section 3.1. For the dynamic solving history, we include (1) what solvers have already been attempted, and (2) the elapsed time since the start of solving.

- **Actions** A: Each $a \in A$ is defined as a pair (v, t), where v is a selected verifier from V and t is an assigned internal time limit chosen from several prefixed options. Therefore, A is a discrete action space.

- **Dynamic Function** D **and Reward** R: The dynamic function $D(s, a) = (s', r)$ deterministically returns both the next state s' and the reward r after taking action $a = (v, t)$ in state s. This state transition reflects the results of executing v with t on the corresponding instance. Our reward r is designed to encourage successfully solving the instance, with a preference for achieving this in a shorter time. Specifically, at each step, r is defined as below:

$$r = \begin{cases} 1 - t_{actual}/T, & \text{if the instance is successfully solved} \\ -t_{actual}/T, & \text{otherwise} \end{cases}$$

  where, t_{actual} is the actual execution time of v. t_{actual} may be equal to t, if v does not return in the assigned t, or t_{actual} may be less than t when v terminates before the assigned time limit. This design of r restrains r between -1 and 1, a common design for RL. r is 1 when the instance is solved instantly, and 0 when the instance is solved using all of T. If the episode fails, i.e., the instance is not solved in T, r is -1. This design shares the same principle as the PAR-2 score, a measure that counts the actual used time for successful instances, while penalizing double the timeout for failed instances.

3.3 Deep Reinforcement Learning

We apply the DQN algorithm (Mnih et al., 2015) to train a policy for this algorithm selection and scheduling problem. A deep neural network $Q(s, a; \theta)$ is used to approximate the Q-value function $Q^*(s, a)$, which estimates the expected cumulative reward for taking action a in state s and following the policy thereafter. The network is trained through interactions with the environment by updating weights to minimize the difference between predicted Q-values and target Q-values, defined by the Bellman equation:

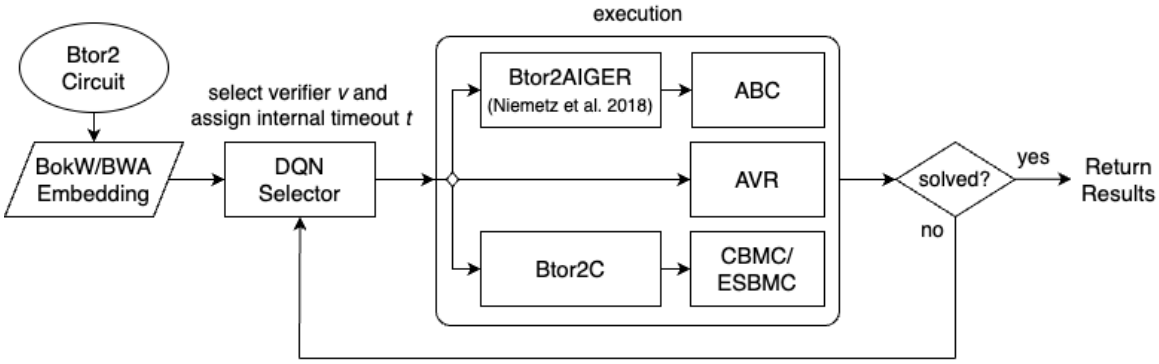

Figure 2: Adaptive Btor2-Select architecture

$$Q(s,a) \leftarrow r + \max_{a'} Q(s',a') \tag{2}$$

where s' is the next state after taking action a.

The training process involves collecting experience tuples (s, a, r, s') and storing them in a replay buffer. These tuples are collected from sampling solving episodes on instances drawn from the distribution D_b. Periodically, the agent samples batches of experience tuples from the replay buffer to update θ according to Equation 2. Through iterative updates and training on experience tuples, $Q(s, a; \theta)$ is expected to approximate the optimal Q-value function $Q^*(s, a)$.

It is interesting to note that, if we restrict ourselves to committing to a single solver without scheduling a sequence, the Q-value function acts as an EHM, predicting the performance of each solver upon a given instance. Therefore, in our DQN setting, the Q-value function can be viewed as an extension of the EHM, adapted to the scheduling scenario.

3.4 Architecture

Figure 2 shows the architecture of our adaptive algorithm selector Btor2-Select. Given an input instance, the selector selects a component verifier v and assigns an internal timeout t for its execution. If the previous attempt is not successful, the selector iteratively makes this choice at each step. This architecture applies to both training and inference time.

We include four model checkers with various configurations in our verifier set. Two hardware model checkers `AVR` (Goel & Sakallah, 2020) and `ABC` (Brayton & Mishchenko, 2010) are the winners of the HWMCC'20 competition; software verifiers `CBMC` (Clarke et al., 2004) and `ESBMC` (Gadelha et al., 2018) showed strong performance on Btor2 instances in our previous work (Beyer et al., 2023). A configuration of the model checker refers to the implementation of a specific algorithm, including property directed reachability (PDR) (Bradley, 2011), bounded model checking (BMC) (Biere et al., 1999), interpolating model checking (IMC) (McMillan, 2003), and k-induction (K-Ind) (Sheeran et al., 2000). See a more detailed description of our component verifiers in Section 4.2.

Verifier	#Solved	PAR-2 (sec)
ABC.BMC	373	1932691.0
ABC.IMC	690	1382558.4
ABC.PDR	**961**	**909922.1**
AVR.BMC	377	1934160.3
AVR.K-Ind	571	1577432.5
AVR.PDR	784	1219703.5
CBMC.BMC	440	1821186.9
CBMC.K-Ind	664	1431101.6
ESBMC.BMC	426	1850108.2
ESBMC.K-Ind	515	1685649.6

Table 1: List of component verifiers and their performance on all 1 441 benchmarks.

4. Experiment Design

We evaluated Btor2-Select on extensive Btor2 benchmarks. This section documents the experiment design while Section 5 presents the results.

4.1 Benchmark Dataset

We have built a comprehensive Btor2 benchmark set at https://gitlab.com/sosy-lab/research/data/word-level-hwmc-benchmarks/ from HWMCC as well as other sources. At this stage, we only focus on the 1 441 benchmarks that do not include array reasoning. Our experiments were conducted over this benchmark set.

4.2 Component Verifiers and their Performance

As mentioned in Section 3.4, we included four model checkers with different configurations in our verifier candidate set V. The model checkers are `ABC` version 1.01, `AVR` version 2.1, `CBMC` version 5.95.1, and `ESBMC` version 7.4.0. Table 1 lists all component verifiers with their performance on all 1 441 benchmarks. The performance measurement of all verifier-instance pairs was executed on Ubuntu 22.04 machines, each with a 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units and 33 GB of RAM. Each task was assigned 2 CPU cores, 15 GB RAM, and 15 min of CPU time limit. We used the BenchExec framework (Beyer, 2016) to ensure reliable resource measurement and reproducible results.

4.3 Adaptive Btor2-Select and Baselines

We implemented our DQN-based adaptive algorithm selector Btor2-Select in Python. In our experiments, the external timeout T was set to 900 seconds. We configured the internal timeout t options to be 100, 300, and 900 seconds. We modeled our MDP problem in Gymnasium (Towers et al., 2024) and used the Stable-Baseline3 (Raffin et al., 2021) implementation for DQN. We included two common algorithm selection methods, i.e., EHM and PWC as baselines. We used XGBoost (Chen & Guestrin, 2016) as the runtime regressor for EHM and the verifier ranker for PWC.

Verifier	#Solved	PAR-2 (sec)	VBS-SBS Closed
EHM-BoKW	211.6	147 632.2	54.82%
EHM-BWA	212.4	146 383.0	57.50%
PWC-BoKW	215.0	140 617.7	66.56%
PWC-BWA	214.0	142 287.7	64.41%
DQN-BoKW	**217.4**	**137 577.0**	**71.10%**
SBS (`ABC-PDR`)	192.2	181 984.4	-
VBS	227.0	119 526.3	-

Table 2: Testing results (average from 5-fold cross-validation) of various Btor2-Select models. The closed gap between VBS and SBS was measured in PAR-2 performance.

4.4 Training and Testing

We evaluated all algorithm selection methods through a training-testing scheme, i.e., the underlying ML models were trained on solving experiences over a training benchmark set, and the trained algorithm selectors were evaluated over a separate testing benchmark set. To ensure robust evaluation, we used 5-fold cross-validation. In this process, the benchmark set was split into five subsets; each subset was used once as the testing set while the others formed the training set. The results from all five iterations were averaged to assess overall performance. This cross-validation evaluation scheme is widely used in algorithm selection literatures (O'Mahony et al., 2008; Scott et al., 2023; Xu et al., 2012).

5. Experiment Results and Analysis

The test results from 5-fold cross-validation are shown in Table 2. The number of test instances was either 288 or 289. One key observation is that all algorithm-selection models effectively improved performance compared to individual tools. The best among non-scheduling methods was the PWC model with BoKW embedding. It closed 66.6% of the VBS-SBS gap. Notably, the DQN model with BoKW embedding, with its power of adaptive scheduling, achieved the best performance among algorithm selectors, closing 71.1% of the VBS-SBS gap. It solved 13.1% more instances and reduced PAR-2 by 24.4% compared to the SBS, `ABC-PDR`.

Measuring Contributions of Component Verifiers Algorithm selection showed strong results on Btor2 problems. Based on these results, we were also interested to see how each component verifier contributed to the collective performance, and especially, how beneficial was to include the software tools in the portfolio. Therefore, following the approach of Fréchette et al. (2016), we measured the Shapely contribution ϕ_{v_n} of each component verifier $v_i \in V$ to the PWC-BoKW portfolio solver, as shown in equation 1. In our case, p maps C, a subset of V, to the number of instances solved by the algorithm selector built from C.

The results are shown in Figure 3. Leveraging the additivity property of the Shapley value, we evaluated the collective contribution of all software verifiers by summing the Shapley values of the individual ones. Overall, the software components contributed 27.2% to the performance of the PWC-BoKW solver.

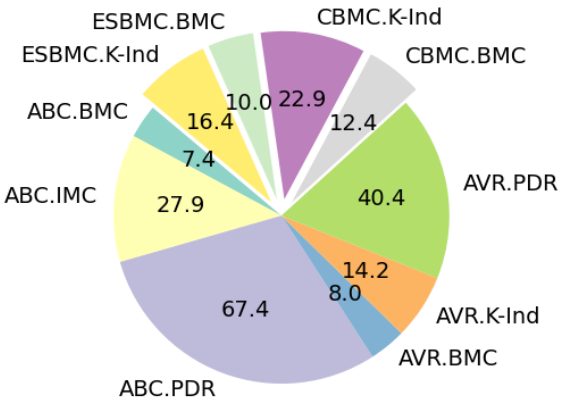

Figure 3: Each component verifier's Shapley-value contribution to PWC-BoKW.

6. HWMCC'24 Submission

We submitted the PWC-BoKW version of Btor2-Select to the HWMCC'24 competition. See the codes and more detailed description of this submission at the repos for the submitted compositional solver and algorithm selector.

7. Future Work

Some to-dos and future directions are outlined below:

- We are implementing the graph-kernel representation (Richter et al., 2020) for Btor2.
- We are investigating the explainability of our algorithm selector. We have done some explorations using the SHAP framework (Lundberg & Lee, 2017).
- More dynamic runtime information can be included in our state representation to help the adaptive selection.

References

A. Biere, N. Froleyks, & M. Preiner. (2020). 11th Hardware Model Checking Competition (HWMCC 2020) [Accessed: 2024-09-13].

Beyer, D. (2016). Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In M. Chechik & J. Raskin (Eds.), *Tools and algorithms for the construction and analysis of systems - 22nd international conference, TACAS 2016, held as part of the european joint conferences on theory and practice of software, ETAPS 2016, eindhoven, the netherlands, april 2-8, 2016, proceedings* (pp. 887–904, Vol. 9636). Springer. https://doi.org/10.1007/978-3-662-49674-9_55

Beyer, D., Chien, P.-C., & Lee, N.-Z. (2023). Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-c translator. *TACAS 2023*, 152–172.

Biere, A., Cimatti, A., Clarke, E., & Zhu, Y. (1999). Symbolic model checking without bdds. *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*, 193–207.

Bradley, A. R. (2011). SAT-based model checking without unrolling. *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 70–87.

Brayton, R., & Mishchenko, A. (2010). Abc: An academic industrial-strength verification tool. *CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, 24–40.

Cenamor, I., de la Rosa, T., Fernández, F., et al. (2014). Ibacop and ibacop2 planner. *IPC 2014 planner abstracts*, 35–38.

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *ACM SIGKDD 2016*, 785–794.

Clarke, E., Kroening, D., & Lerda, F. (2004). A tool for checking ANSI-C programs. *TACAS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*, 168–176.

Fréchette, A., Kotthoff, L., Michalak, T., Rahwan, T., Hoos, H., & Leyton-Brown, K. (2016). Using the shapley value to analyze algorithm portfolios. *AAAI 2016*, *30*.

Gadelha, M. R., Monteiro, F. R., Morse, J., Cordeiro, L. C., Fischer, B., & Nicole, D. A. (2018). Esbmc 5.0: An industrial-strength c model checker. *ACE 2024*, 888–891.

Goel, A., & Sakallah, K. (2020). Avr: Abstractly verifying reachability. *TACAS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part I 26*, 413–422.

Le Berre, D., Roussel, O., & Simon, L. (2024a). SAT competition 2007 [Accessed: 2024-10-27]. https://satcompetition.github.io/

Le Berre, D., Roussel, O., & Simon, L. (2024b). SAT competition 2009 [Accessed: 2024-10-27]. https://satcompetition.github.io/

Leeson, W., & Dwyer, M. B. (2024). Algorithm selection for software verification using graph neural networks. *ACM Transactions on Software Engineering and Methodology*, *33*(3), 1–36.

Leyton-Brown, K., Nudelman, E., & Shoham, Y. (2002). Learning the empirical hardness of optimization problems: The case of combinatorial auctions. *Principles and Practice of Constraint Programming-CP 2002: 8th International Conference, CP 2002 Ithaca, NY, USA, September 9–13, 2002 Proceedings 8*, 556–572.

Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems 30* (pp. 4765–4774). Curran Associates, Inc. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

McMillan, K. L. (2003). Interpolation and SAT-based model checking. *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*, 1–13.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Niemetz, A., Preiner, M., Wolf, C., & Biere, A. (2018). Btor2, btormc and boolector 3.0. *CAV 2018*, 587–595.

O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., & O'Sullivan, B. (2008). Using case-based reasoning in an algorithm portfolio for constraint solving. *Irish conference on artificial intelligence and cognitive science*, 210–216.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *J. Mach. Learn. Res.*, *22*, 268:1–268:8. https://jmlr.org/papers/v22/20-1364.html

Rice, J. R. (1976). The algorithm selection problem. In *Advances in computers* (pp. 65–118, Vol. 15). Elsevier.

Richter, C., Hüllermeier, E., Jakobs, M.-C., & Wehrheim, H. (2020). Algorithm selection for software validation based on graph kernels. *Automated Software Engineering*, *27*(1), 153–186.

Richter, C., & Wehrheim, H. (2020). Attend and represent: A novel view on algorithm selection for software verification. *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 1016–1028.

Scott, J., Niemetz, A., Preiner, M., Nejati, S., & Ganesh, V. (2023). Algorithm selection for SMT: MachSMT: Machine learning driven algorithm selection for SMT solvers. *International Journal on Software Tools for Technology Transfer*, *25*(2), 219–239.

Shapley, L. S. (1953). A value for n-person games. *Contributions to the Theory of Games*, *2*.

Sheeran, M., Singh, S., & Stålmarck, G. (2000). Checking safety properties using induction and a SAT-solver. *International conference on formal methods in computer-aided design*, 127–144.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, *abs/1712.01815*. http://arxiv.org/abs/1712.01815

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Towers, M., Kwiatkowski, A., Terry, J. K., Balis, J. U., Cola, G. D., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, H., & Younis, O. G. (2024). Gymnasium: A standard interface for reinforcement learning environments. *CoRR*, *abs/2407.17032*. https://doi.org/10.48550/ARXIV.2407.17032

Tulsian, V., Kanade, A., Kumar, R., Lal, A., & Nori, A. V. (2014). Mux: Algorithm selection for software model checkers. *Proceedings of the 11th Working Conference on Mining Software Repositories*, 132–141.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2012). Evaluating component solver contributions to portfolio-based algorithm selectors. *SAT 2012*, 228–241.

Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, *32*, 565–606. https://doi.org/10.1613/jair.2490