Analysing ProB's Constraint Solving Backends

What Do They Know? Do They Know Things? Let's Find Out!

Jannik Dunkelau(✉), Joshua Schmidt, and Michael Leuschel

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, Universitätsstraße 1, 40225 Düsseldorf, Germany
{jannik.dunkelau,joshua.schmidt,michael.leuschel}@hhu.de

Abstract. We evaluate the strengths and weaknesses of different backends of the ProB constraint solver. For this, we train a random forest over a database of constraints to classify whether a backend is able to find a solution within a given amount of time or answers *unknown*. The forest is then analysed in regards of feature importances to determine subsets of the B language in which the respective backends excel or lack for performance. The results are compared to our initial assumptions over each backend's performance in these subsets based on personal experiences. While we do employ classifiers, we do not aim for a good predictor, but are rather interested in analysis of the classifier's learned knowledge over the utilised B constraints. The aim is to strengthen our knowledge of the different tools at hand by finding subsets of the B language in which a backend performs better than others.

Keywords: Constraint solving · Machine learning · Decision trees · Feature importances · Association rules · Automated tool selection

1 Introduction

Besides its native CLP(FD)-based backend, the validation tool ProB [30] offers various backends for solving constraints, e.g. encountered during symbolic verification. In previous work [18,19], we trained neural networks to decide for a given constraint which backend should be used. We compared two approaches: one based on feature vectors derived from domain knowledge, and one based on encoding constraints as images. While we achieved promising results with the image-based approach, it was not possible to extract a comprehensible explanation about how the predictions were made. In follow-up work [34] the experiment was replicated with decision trees [5] using the same feature sets as before. This was motivated by the fact that decision trees are a transparent machine learning algorithm allowing to extract and interpret the learned decision rules and thus the acquired knowledge.

A. Raschke et al. (Eds.): ABZ 2020, LNCS 12071, pp. 107–123, 2020.
https://doi.org/10.1007/978-3-030-48077-6_8

In this paper we will expand on the decision tree approach and further analyse the relative importances of the features used for deciding whether the different backends of ProB will be successful or not. Moreover, we will compare these results with our *a priori* assumptions about the subdomains in which each backend should work well. While we will display achievable classification performances for our predictors, we do not aim for a good performance, but instead for an analysis over the whole dataset. In particular, we are not interested in replacing the decision function in ProB with a predictor presented in this paper. The goal is to find subsets of the B language in which a backend performs better than others to strengthen our knowledge of the different tools at hand. With the gathered information we may be able to improve the ProB constraint solver and to obtain more suitable features sets for related machine learning tasks for B in the future.

2 Primer on ProB and its Backends

ProB [29,30] is an animator, model checker, and constraint solver for the formal specification language B [1]. The B language allows to specify, design, and code software systems as well as to perform formal proof of their properties. When using ProB, properties can be checked exhaustively on a state space using various model checking techniques. B is rooted in predicate logic with arithmetic and set theory. At the heart of ProB is a constraint solver for the B language. ProB's constraint solver is used for many tasks. During animation it has to find suitable parameters for the B operations and compute the effect of executing an operation, during disproving [26] it is used to find counter examples to proof obligations. The constraint solver is also used for test case generation, symbolic model checking or program synthesis.

ProB has actually not one but three constraint solving backends and each backend has a variety of options. In Sects. 2.1, 2.2 and 2.3 we will introduce each backend, outline their differences, and summarise our a priori assumptions about their performances on subdomains of the B language.

2.1 The Native CLP(FD) Backend

ProB's kernel [29] is implemented in SICStus Prolog [11] using features such as co-routines for delayed constraint propagation, or mutable variables for its constraint store. The CLP(FD) finite domain library [10] is used for integers and enumerated set elements. The library has a limited precision of 59 bits. ProB handles overflows by custom implementations and also supports unbounded domains as well as symbolic representations for infinite or large sets. Some specific features of the ProB constraint solver are that it computes all solutions to a constraint using backtracking. This is important as constraints are often used within set comprehensions. It is also important for model checking to ensure that the entire state space is constructed. ProB can deal with higher-order sets, relations and functions.

Subdomains in Which CLP(FD) Presumably Performs Better. First and foremost, the CLP(FD) backend of ProB is the only backend supporting all constructs available in B. It is thus the default backend. It performs best for constraints arising in animation, where usually a small number of variables (operation parameters) have to be enumerated. In this context, it can deal well with large data values.

Generally speaking, ProB performs well on constraints using enumerated sets, booleans and/or bounded integers as base types. It performs reasonably well on unbounded intervals if interval reasoning can be applied. While ProB is very good at model finding, it can only detect unsatisfiability by exhaustively enumerating all values remaining after deterministic propagation. In case of unbounded data structures, ProB cannot exhaustively enumerate all cases and is much less powerful. While CLP(FD) cannot natively handle unbounded domains or quantifiers, ProB's backend contains several custom extensions to do so. A key limitation of the CLP(FD) backend is that it has no features such as backjumping, conflict-driven clause learning, or random restarts. In consequence, the backend can get stuck in the search space repeatedly enumerating invalid values which SAT or SMT solvers would rule out by learning.

2.2 The Kodkod Backend

An alternative backend [35] for ProB makes use of Alloy's Kodkod library [38] to translate constraints to propositional logic, which are then solved by a SAT solver. For instance, sets are translated as bit vectors. In particular, a subset x of the interval 0..2 would be translated into three propositional logic variables x_0, x_1, x_2 where x_i is true if $i \in x$ holds. The constraint $\{1, 2\} \subseteq x$ can then be translated to the propositional logic formula $x_1 \wedge x_2$. As Kodkod does not allow higher-order values, any such constraint is not passed to Kodkod and is instead dealt with by ProB's default CLP(FD) backend after Kodkod has found a solution for the other constraints.

When using this backend, ProB will first perform an interval analysis and determine which variables have a finite scope and a first-order type. The constraint is then partitioned into a part sent to Kodkod and a part solved by ProB. During solving, the SAT solver is called first. For every solution obtained by the SAT solver, ProB's CLP(FD) backend solves the remaining constraints. By default, Kodkod's Sat4j [28] SAT solver is selected.

Subdomains in Which Kodkod Presumably Performs Better. The strengths and weaknesses of the backend based on Kodkod stem from its internal reliance on SAT solving. While modern SAT solvers are very fast when it comes to solving very large boolean formulae, encoding B into propositional logic underlies certain restrictions. SAT encodings can only be used for data types known to be finite. In particular, one has to assign an upper and lower bound for integers and set sizes. Thus, integer overflows might occur and it is hard to ensure soundness and completeness. Furthermore, arithmetic operations have to be encoded in propositional logic as well such as binary adders. This leads to additional overhead when generating a conjunctive normal form, especially for large bit widths. The designers of Alloy argue [25] that lack of integers is not disadvantageous in general, as integer constraints are often of secondary nature. In B models, this is not the case. In summary, this backend is not good for arithmetic, large relations, infinite domains, higher-order constraints, or data structures.

In contrast, SAT solving is ideal for problems involving relations as those can be expressed in a way suitable for Kodkod's backends [35]. Furthermore, given that Kodkod is originally used as a backend for analysing Alloy it has been tuned towards constraints involving operations on relations. For instance, the relational image or transitive closure operations of B are handled efficiently by the translation to SAT using Kodkod.

2.3 The Z3 Backend

The third backend of ProB translates B constraints to SMT-LIB formulae and targets the SMT solvers Z3 [13] and CVC4 [3]. Here we focus on the Z3 binding [27] only. The translation works by rewriting the B constraints into a normal form using a core subset of the B operators which can be mapped to SMT-LIB. Additional variables, set comprehensions, and quantifiers are introduced for those operators which have no counterpart in SMT-LIB or Z3, e.g. cardinality, or minimum and maximum of an integer set. Functions and relations are translated to the Array theory of SMT-LIB. The DPLL(T) [21] algorithm underlying SMT solvers is fundamentally different from CLP(FD). Just like for the SAT translation, SMT solvers can perform backjumping and conflict-driven clause learning.

Subdomains in Which Z3 Presumably Performs Better. SMT solvers such as Z3 are very good at proof for B and Event-B (cf. [14,15]). Our experience in the context of model finding is that Z3 is good at detecting inconsistencies, in particular on infinite domains. For example, Z3 is able to detect that the constraint $x < y \wedge y < x$ is unsatisfiable. The other two backends are unable to detect this using their default settings. Note that ProB is able to detect this inconsistency if one enables an additional set of propagation rules based on CHR.

On the downside, Z3 often has difficulties to deal with quantifiers. Moreover, the translation from B to SMT-LIB does not yet support various operators such as general union or general sum nor does it support iteration and closure operators. Constraints using one of these operators are not translated to SMT-LIB at all and the backend returns *unknown*. In summary, the Z3 backend is good at detecting inconsistencies and reasoning over infinite domains, but for constraints involving quantifiers, larger data values or cardinality computations it often answers *unknown*.

3 Primer on Decision Trees and Random Forests

We utilise techniques of supervised machine learning to train a classifier for B constraints, which we will then further analyse in Sect. 6.

The notion of machine learning covers a family of algorithms which are able to improve their predictions using a dataset processed at a so called *training* time. For *supervised* machine learning, this dataset consists of tuples $(x, y) \in D \subseteq X \times Y$, where $x \in X$ represents the input data and $y \in Y$ is the corresponding *ground truth*, which is the correct class label to be predicted by the employed algorithm. For instance, for a binary classification task, the ground truth can be either 0 or 1. Usually, $X = \mathbb{R}^d$ corresponds to a d-dimensional feature space, where each problem instance to be classified is represented as a feature vector $x = \langle x_1, \ldots, x_d \rangle$. Each x_i hereby refers to a specific characterisation, i.e. *feature*, of the problem instance. During training time, the algorithm is supposed to learn the mapping $x \mapsto y$ for each $(x, y) \in D$ by generalising over recurring patterns in the input data X. It is important that this learned mapping is accurate yet as general as possible, so as to cover yet unseen problem instances. A classifier is said to *overfit* on the training data if its performance in classifying unseen data is significantly worse. To detect possible overfitting, the resulting classifier's performance is evaluated on a separate *test set*, i.e. a data set which was not experienced during training time.

In this article, we employed decision trees as the machine learning algorithm of choice. They correspond to a supervised learning method where the training data at the root of the decision tree is progressively split into smaller subsets using a feature-based splitting criterion. At the leaves of the decision tree only subsets with the same ground truth remain. Such subsets are referred to as pure subsets.

A variety of splitting criteria exist. For example, the CART algorithm [5] is based on the *Gini impurity* $i(t)$ [31] of a node t defined as

$$i(t) = 1 - \sum_{c \in C} p_c(t)^2$$

with C being the set of possible classes, and $p_c(t)$ is the relative frequency of the elements in t belonging to the class c. For a pure subset t' of a class c, $p_c(t')$ will be 0 for $c \neq c'$ and 1 for $c = c'$. Hence $i(t') = 0$. For an evenly distributed node t'' we have $i(t'') = 1 - \frac{1}{|C|}$, where $|C|$ is the cardinality of C.

The goal of the decision tree learning algorithm is to reach an impurity of 0 with as few splits as possible. For any split of t into two sub-nodes t_L and t_R, we thus measure the *impurity decrease* by

$$d(t_{(L,R)}) = i(t) - i(t_L)\frac{|t_L|}{|t|} - i(t_R)\frac{|t_R|}{|t|}\ .$$

The split which maximises the impurity decrease is finally chosen and the algorithm is called recursively on t_L and t_R respectively. A decision tree is shown in Fig. 1, where leaves represent actual classes.

3.1 Random Forests

Random forests [7] are a bagging approach [6] to decision trees, i.e. instead of only training a single decision tree, a set of k decision trees $(T_i)_{1 \leq i \leq k}$ is trained.

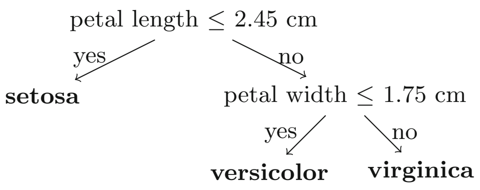

Fig. 1. Decision tree classifier over a set of iris flowers [20]. The species iris setosa, iris versicolor, or iris virginica is classified based on petal length and width.

Each tree is trained on a random subset of the training samples as well as a random subset of features. This randomisation ensures most trees in the set to be distinct from each other. For example, the impurity decrease of common features will vary between the training samples, leading to different choices of splitting. Due to bagging, the relatively unstable nature of decision trees is countered and the technique is less prone to overfitting.

A measure for the relative importances of each feature in a random forest is the *mean decrease importance* [7]. The mean decrease importance of a feature averages the impurity decrease per feature over each decision tree in the forest. Hence, it is a measure of the average impurity decrease the feature offers [2,37].

3.2 Rationale for Using Random Forests

While we had multiple classification algorithms to choose from, we finally settled on random forests. This choice was motivated by our need for a strong and interpretable classifier.

In previous work [19] we used convolutional neural networks, but we were unable to extract the knowledge accumulated by the classifciation due to the black box nature of the neural networks. Hence, we started to use decision trees, as one can easily extract classification rules after the training phase. These rules are comprehensible and can be interpreted by non-experts as well. Decision trees also offer insights about the relevancy of features: the closer to the root a split over a specific feature is done, the more impact it has for the decision process.

Alternate machine learning approaches are linear regression and clustering approaches. For linear regression the relevance of features could be extracted by examining the relative differences in their coefficients. However, this would not yield direct rules describing *why* a particular prediction was made. As we are particularly interested in extractable knowledge from trained classifiers and reasoning for the given predictions, we favoured decision trees over linear regression. On a similar note, we decided against clustering. However, a clustering approach for grouping similar constraints together presents an interesting alternative approach to be studied in future work.

In the end, we decided to utilise random forests for the present article. Although they are again blackbox algorithms, they consist of interpretable pieces, which can be analysed for more general rules [16,23].

4 Related Work

The related work in the field is split into two categories: machine learning powered algorithm portfolios for SMT solving, and knowledge extraction from tree ensemble learners such as random forests. To the best of our knowledge, no intersection of both categories exists yet in literature, as we do in this article.

Healy et al. [24] conducted a solver portfolio for the Why3 platform [4]. The solver selection was done via decision trees which predicted the anticipated runtime of a proof obligation for each solver, and choosing the fastest one. James P. Bridge [8] used support vector machines for automating the heuristic selection for the E theorem solver [36]. While he was able to improve the already implemented auto-mode in E, he also investigated picking a minimal feature set which ultimately consisted of only two to three features.

Yang et al. [39] analysed decision trees to extract minimal feature subsets which need to be flipped to achieve a more favourable outcome. Their application area was customer relationship management with focus on increasing the amount of loyal customers, i.e. detect what needs to be done to turn a regular customer into a loyal customer. Similarly, Cui et al. [12] proposed an integer linear program on random forests for finding the minimal subset of features to change for obtaining a different classification. Deng [16] proposed interpretable trees (inTrees) for interpreting tree ensembles. In their paper, they propose a set of metrics to extract learned knowledge from a tree ensemble such as a random forest. This includes the actual rules learned in an ensemble as well as frequent variable interactions. Narayanan et al. [32] extracted the most common patterns for failing solid state drives in datacenters using inTrees. In their work, they found that these extracted patterns match with previously made observations.

5 Experimental Setup

In this section, we briefly outline the training data and the feature set in use.

5.1 The Training Data

For acquiring the constraints for the training data, we extracted B predicates from the public ProB examples repository[1] and constructed more complex constraints inspired by ProB's enabling analysis [17] or discharging proof obligations [26]. Each backend was given a timeout of 25 s to decide whether the constraint has a solution or is a contradiction. Constraints for which a definite answer was found build up the positive class for a solver. The negative class is made up of the other outcomes: timeouts, errors, or the answer *unknown*. Overall, the class distribution was imbalanced, as for instance only about 35% of samples belonged to the negative class for the CLP(FD) backend. Yet, we do not deem this as a problem because the decision trees are trained with respect to a weighted training set.

[1] https://www3.hhu.de/stups/downloads/prob/source/ProB_public_examples.tgz.

The choice of the 25 s timeout was arbitrary. However, we evaluated how much more constraints are assigned to the positive class compared to using ProB's default timeout of 2.5 s. The CLP(FD) backend is able to solve 65.47% of the constraints using a timeout of 2.5 s, while the Kodkod and Z3 backends solve 64.65% and 21.52% respectively. When increasing the timeout by factor 10 to 25 s, these percentages increase to 65.48% for CLP(FD) (+0.01%), 64.67% for Kodkod (+0.02%), and 21.53% for Z3 (+0.01%). As the percentage of solvable constraints for each backend only increased by a rather insignificant amount, we deemed the unsolvable constraints as complex enough for our analysis approach. We did not test with higher timeouts.

For each backend's analysis we had around 170,000 unique samples.

5.2 The Feature Set

For training the decision trees, we created a manually selected set of 109 features (further referred to as F109) which mainly consists of characteristics such as the amount of arithmetic operations per top level conjunct, or the ratio of intersections of all used set operators. Further features consist of maximum and mean nesting depths for certain language constructs such as negations and powersets, or the amount of unique identifiers per top level conjunct and number of interactions between them. Additionally, identifiers are grouped into unbounded, semi-bounded (only upper or lower bound), and fully bounded (both, upper and lower bound) identifiers. This grouping is sensitive to whether the boundaries are explicitly set (e.g. `a < 5`) or only bounded by another identifier (`a < b`).

As we are interested in the knowledge gathered by the random forests over the whole corpus of B constraints at our disposal, we will not split the dataset into sets for training and testing for our final analysis as is common for classification tasks aiming for a good predictor. However, as a sanity check that the selected features are indeed discriminatory enough to actually learn weaknesses and strengths of each backend, we still analysed the predictive performances of a random forest for each backend on a classical split into datasets for training and testing. For measuring performance, we utilised the metrics *accuracy*, *balanced accuracy* [9], and the F_1*-score* [22].

Each prediction of a classifier can either be a true positive (tp), true negative (tn), false positive (fp) or false negative (fn), i.e. the prediction can be either correct or false corresponding to either the positive or negative classes 1 and 0. The utilised performance metrics are defined as follows:

$$\textit{accuracy} = \frac{\mathrm{tp} + \mathrm{tn}}{\mathrm{tp} + \mathrm{tn} + \mathrm{fp} + \mathrm{fn}},$$

$$\textit{balanced acc.} = \frac{1}{2}\left[\frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fn}} + \frac{\mathrm{tn}}{\mathrm{tn} + \mathrm{fp}}\right].$$

Accuracy describes the percentage of the test data which were classified correctly. Balanced accuracy is most suitable for an unbalanced dataset in which the distribution of classes is not equal. It averages the percentage of correctly

Table 1. Random Forest classification performances over the set of 109 features.

Backend	Dataset	Accuracy	Balanced acc.	F_1-score
CLP(FD)	F109	0.947	0.926	0.966
Kodkod	F109	0.926	0.906	0.950
Z3	F109	0.919	0.873	0.797

predicted samples per class. The F_1-score is defined as the harmonic mean over the notions *precision* and *recall* [22]:

$$precision = \frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fp}}, \quad recall = \frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fn}}$$

$$\mathrm{F}_1 = 2 * \frac{precision * recall}{precision + recall}\ .$$

Precision describes the probability of a positive prediction to be correct. Recall describes the probability for samples of the positive class to be classified as such.

Table 1 shows the results of this sanity check. We used 80% of the data for training, whereas the performance measures were taken on the remaining 20%. Each classification task was concerned with whether a backend would return a definitive answer for a given constraint (satisfiable or unsatisfiable) or would yield *unknown*. As the performance scores are all higher than 0.9, we deem the feature set F109 to be suitable for our purposes.

6 Analysis and Results

For each backend, we trained a random forest with 50 trees using the Gini impurity decrease splitting criterion to predict whether the respective backend can find an answer or leads to *unknown*. As machine learning framework, we employed the Scikit-learn Python library [33].

For the following analysis, we utilised the whole dataset of 109 features (F109) as the training set and did not make use of a test set, as we are more interested in an analysis of random forests containing information of all the data.

Please note that this work only considered the default settings of each backend. It is relevant to mention that multiple settings exist which could influence the respective outcomes of the analysis. For instance, although the CLP(FD) backend has problems with detecting inconsistencies over unbounded domains such as $x < y \wedge y < x$, one can activate an additional CHR propagation which improves detection of inconsistencies in general as mentioned in Sect. 2.3.

6.1 Feature Importances

In order to gain a deeper insight in the feature set we compute the *Gini importance* which is the mean decrease importance of a feature within a random forest using the Gini impurity as the splitting criterion.

Table 2. Top ten features for each backend ranked by the Gini importance.

Backend	Most important features (descending)
CLP(FD)	*Function application*, *max conjunct depth*, *forward compositions*, *relational overrides*, *nested logic with conj.*, *nested logic with implications*, equalities, *function variables*, subset ratio, *identifier count*
Kodkod	*Function application*, *function vars*, *forward compositions*, set op., *nested logic with conj.*, *nested logic with disj.*, nested logic with impl., *avg. powerset nesting*, *identifier count*, *relational overrides*
Z3	Relational operators, domain ops, functions, *function vars*, *avg. powerset nesting*, domain restrictions, unbounded domains, *identifier count*, *max. conjunct depth*, *function application*

Table 2 shows the top ten features that are necessary to classify the data at hand for each backend. The common features of the three subsets are *highlighted*. These indicate a particularly high importance as they are used for each of the three backends' decisions.

The CLP(FD) backend and the Kodkod backend have the most features in common. The most important feature for both backends is the presence of function applications. Indeed, a function application is a complex operation for a constraint solver since it entails for example the well-definedness condition that the applied value is an element of the function's domain. Both backends' classifiers favour the presence of nested logic formulae with further possibly nested conjunctions, disjunctions, and implications, indicating more involved constraint as well. The initial assumption that the Kodkod backend is better suited to solve constraints over relations is strengthened by the high ranking of the ratio of relational compositions and overrides in the top 10 features. Of course, the overall higher similarity of the top ranked features for the CLP(FD) and Kodkod backend is influenced by the fact that constraints that cannot be translated to SAT are solved by ProB.

The gathered feature set for the classifier of the Z3 backend favours the presence of relational operations, in particular, the presence of domain operations. As initially expected, the feature representing the presence of unbounded domains has a high importance as well.

While this analysis allows for selection of features for the sole purpose of classification, it does not yet give us info as of why a feature ranks high. For instance, it remains uncertain whether the presence of relational operators correlates to Z3's positive or negative class. This will be analysed in Sect. 6.2.

Classifying on Reduced Feature Sets. While we are mostly interested in analysing which language subsets are hard for a backend to solve, we can evaluate the significance of the most relevant features (determined via Gini relevance as done above) by conducting a regular classification over only these relevant features. When using the ranked feature sets to find a minimal set of features, we have to consider that at least one feature exists for each B data type or group of operations, e.g. relational operators, as the dataset might be biased to specific operations. For instance, the fact that the presence of arithmetic operations is not ranked high does not mean there should be no such feature at all in general.

Table 3. Random forest classification performances for minimised feature sets.

Backend	Dataset	Accuracy	Balanced acc.	F_1-score
CLP(FD)	F10	0.914	0.875	0.944
	F50	0.947	0.929	0.966
Kodkod	F10	0.887	0.853	0.923
	F50	0.924	0.907	0.949
Z3	F10	0.875	0.804	0.747
	F50	0.916	0.870	0.795

We created two features sets containing the top 10 and 50 ranked features for each backend, referred to as F10 and F50 respectively. The results presented in Table 3 show that the minimised subsets of 50 features capture the problem domain as well as the larger set containing 109 features presented in Table 1. The smaller subset containing 10 features already shows good performance but does not perform as well as the one using 50 features, indicating that the problem at hand is complicated at least.

6.2 Association Rule Analysis

Our main goal is to determine how the backends perform on different subsets of the B language. For this we performed an association rule analysis using the inTrees framework [16], thereby identifying frequent feature interactions as well as determining those syntax elements which increase the chance of unsolvability for each backend. For the analysis, we interpret paths from the root to the leaves of each decision tree in the forest as a single rule. Each node in these trees corresponds to a feature along with a threshold value for deciding which path to follow. An example based on the decision tree from Fig. 1 is given in Fig. 2.

petal length $(<) \Rightarrow$ setosa
petal length $(>) \wedge$ petal width $(<) \Rightarrow$ versicolor
petal length $(>) \wedge$ petal width $(>) \Rightarrow$ virginica

Fig. 2. Association rules extracted from the decision tree in Fig. 1.

Different paths might be identical up to the respective threshold values. In our analysis, we discard the threshold values and only consider the tendency (below or above threshold) for each rule. This way we can compare rules without having to worry about mismatching threshold values while still accounting for the feature's tendency. Table 4 displays several rules that were collected from the random forest trained for each backend.

Deng [16] uses two metrics for the association rules, *support* and *confidence*. Given two rules $a = \{C_a \Rightarrow Y_a\}$ and $b = \{C_b \Rightarrow Y_b\}$ where C_a, C_b are the respective conditions and Y_a, Y_b the respective outcomes. Rule b is said to be in the support of rule a iff $C_a \subseteq C_b$. That is, each feature used in C_a is also used in C_b (with equal threshold tendency). Let $\sigma(a) = \{r \mid r \text{ is in the support of } a\}$ denote the support set of a. The confidence of an association rule a is then defined as $c(a) = |\{\{C_r \Rightarrow Y_r\} \in \sigma(a) \mid Y_r = Y_a\}|/|\sigma(a)|$, i.e. the ratio of rules in the support of a with the same outcome as a.

For a deeper analysis of the subproblems' performances for each backend, we calculated the support and confidence of the respectively 250,000 shortest rules of the corresponding random forests.

Table 4. Exemplary association rules with their corresponding support and confidence values (Supp. and Conf. respectively). The operators < and > indicate whether the feature value is above or below the learned threshold.

Backend	Rule	Supp.	Conf.
CLP(FD)	Function applications (<) $\wedge$ conjunctions (<) $\wedge$ quantifiers (>) $\wedge$ logic operators (>) $\wedge$ functions (>) $\Longrightarrow$ negative	853	0.69
Kodkod	Function applications (<) $\wedge$ conjunctions (<) $\wedge$ disjunctions (>) $\wedge$ implications (<) $\wedge$ powersets (<) $\wedge$ inequality (>) $\wedge$ quantifiers (<) $\wedge$ lambda-expression ratio (<) $\wedge$ relational inversions (<) $\wedge$ sequences (<) $\wedge$ $\Longrightarrow$ negative	2413	0.79
Z3	Relational operations (<) $\wedge$ functions (<) $\wedge$ unbounded variables (>) $\wedge$ set inclusions (member, subset) (<) $\wedge$ sequences (<) $\wedge$ set operations (<) $\Longrightarrow$ positive	24,479	0.69

Analysis for CLP(FD). For ProB's native backend, most rules with high support only had a confidence of 50%, rendering them insignificant for our analysis. While higher confidence rules had less support such as the one presented in Table 4, they allowed for a look on certain subareas in the problem domain in which the backend struggles to find an answer for.

Main concern for the backend appears to be function applications because they are the most relevant feature for deciding whether the CLP(FD) backend is able to satisfy or reject a constraint according to the analysis in Sect. 6.1.

The implementation of function applications in ProB consists of many special cases such as different treatment for partial or total functions. Moreover, function applications entail a well-definedness condition leading to more involved constraints and possibly weaker propagation. In particular, the constraint solver has to deduce that the values applied to a function are part of its domain which increases complexity drastically if domains are (semi-)unbounded. The multitude of such cases might emphasise the overall complexity for constraint solving and be the reason for function applications leading to negative predictions. This finding suggests the need for a more involved statical analysis of constraints with function types by means of discarding well-definedness constraints early to allow for a more aggressive propagation of function applications. Thus the solver would not need to wait for verification of whether an element actually resides in a function's domain or not.

Further findings show that the use of implications, equivalences, nested powersets as well as operations on powersets contribute to the probability for the backend to answer *unknown* for a given constraint, as do operations concerning multiple variables representing functions and unbounded domains.

Comparing this to our initial presumptions made in Sect. 2.1, the particular difficulty associated with function application was mostly unexpected. Furthermore, while we did not anticipate implications or equivalences to have such significance, their role for unsolvability might be caused by a lot of backtracking inside the constraint solver for satisfying these constraints. The analysis did not bring up further results mismatching our assumptions from Sect. 2.1.

Analysis for Kodkod. The Kodkod backend struggles with arithmetic and powersets, which was to be expected. As already observed with the native backend, we also found an increase in logical operators to increase the constraint complexity significantly. An increase in logical operators naturally increases the nesting depth of the top-level conjuncts, leading to much more involved constraints. The use of functions only appears to be a problem for Kodkod if these are not manipulated by relational operators, rendering Kodkod as a more suitable choice over CLP(FD) in these cases. We generally found our expectations from Sect. 2.2 met regarding Kodkod's handling of relations.

Most positive rules favouring relational operators only showed a small support but had high confidence values and mostly differed in a single feature describing a different relational operator. If one was to generalise these rules into a singular one which is independent of the particular relational operator, these rules should be able to support each other while maintaining their high confidence. This suggests the use of relational operators for the Kodkod backend.

Note again that the Kodkod backend has a fallback to the CLP(FD) backend for non-translated structures, hence both backends perform similar overall.

Analysis for Z3. Contrary to the two backends presented above, the Z3 backend's association rule analysis delivered many high-support/high-confidence rules for the positive class. Table 4 shows one such rule with high support and confidence. Since the analysis did not provide rules with high support and confidence for the negative class, we compared absence of syntax elements in the positive rules to their existence in low-support negative rules for analysis of areas where Z3 does not perform well.

The results suggest that Z3 handles unbounded domains well and favours integer variables and inequality constraints. This is in line with our expectations from Sect. 2.3. However, we observed good performance for relational operators as well which goes against initial presumptions, although this is correlated to the amount of domain restrictions in use. Otherwise, Z3 lacks performance with quantifiers, set comprehensions, powersets, or set operations (as was expected).

The main issues for the Z3 backend are the non-translated operators as well as highly involved translations as outlined in Sect. 2.3. Revisiting these translations and comparing their implementations to those of well-performing syntax elements might allow to increase the backend's performance on further language subsets significantly. For instance, the translation of relational operators might inspire the translation of certain set operators.

7 Conclusion

In this article, we identified subproblems of the B language for which the individual ProB constraint solving backends performed better or worse respectively.

While our findings generally matched our expections stated in Sects. 2.1, 2.2 and 2.3, we found certain results which we did not explicitly expect. For instance, our evidence suggests a difficulty for dealing with function applications as well as implications and equivalences. Involved constraints containing many nested conjunctions and disjunctions also increased the chance for the backends to return *unknown*. Surprisingly, the Z3 backend performed much better on relational operators as expected. As a consequence, our analysis identified the need for a more sophisticated handling of function application and nested logic operators.

As by-product of this work, we were also able to train well-performing classifiers for each backend, which can be used for automated backend selection.

The experimental data as well as corresponding Jupyter notebooks are available on GitHub:

https://github.com/jdnklau/prob-backend-analysis.

Acknowledgements. Computational support and infrastructure was provided by the "Centre for Information and Media Technology" (ZIM) at the University of Düsseldorf (Germany).

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Archer, K.J., Kimes, R.V.: Empirical characterization of random forest variable importance measures. Comput. Stat. Data Anal. **52**(4), 2249–2260 (2008). https://doi.org/10.1016/j.csda.2007.08.015
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland, pp. 53–64, August 2011
5. Breiman, L., Friedman, J., Olshen, R., Stone, C.: Classification and Regression Trees. Wadsworth and Brooks, Monterey (1984)
6. Breiman, L.: Bagging predictors. Mach. Learn. **24**(2), 123–140 (1996). https://doi.org/10.1007/BF00058655
7. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
8. Bridge, J.P.: Machine learning and automated theorem proving. Technical report, University of Cambridge, Computer Laboratory (2010)
9. Brodersen, K.H., Ong, C.S., Stephan, K.E., Buhmann, J.M.: The balanced accuracy and its posterior distribution. In: 2010 International Conference on Pattern Recognition, pp. 3121–3124. IEEE, August 2010. https://doi.org/10.1109/ICPR.2010.764
10. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0033845
11. Carlsson, M., et al.: SICStus Prolog User's Manual, vol. 3. Swedish Institute of Computer Science Kista, Sweden (1988)
12. Cui, Z., Chen, W., He, Y., Chen, Y.: Optimal action extraction for random forests and boosted trees. In: International Conference on Knowledge Discovery and Data Mining KDD 2015, pp. 179–188. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2783258.2783281
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: SMT solvers for Rodin. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 194–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_14
15. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: Integrating SMT solvers in Rodin. Sci. Comput. Program. **94**, 130–143 (2014). https://doi.org/10.1016/j.scico.2014.04.012
16. Deng, H.: Interpreting tree ensembles with intrees. Int. J. Data Sci. Anal. **7**(4), 277–287 (2019). https://doi.org/10.1007/s41060-018-0144-8
17. Dobrikov, I., Leuschel, M.: Enabling analysis for Event-B. Sci. Comput. Program. **158**, 81–99 (2018). https://doi.org/10.1016/j.scico.2017.08.004
18. Dunkelau, J.: Machine learning and AI techniques for automated tool selection for formal methods. In: Proceedings of the PhD Symposium at iFM'18 on Formal Methods: Algorithms, Tools and Applications, University of Oslo, September 2018. https://doi.org/10.18154/RWTH-CONV-236485

19. Dunkelau, J., Krings, S., Schmidt, J.: Automated backend selection for ProB using deep learning. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2019. LNCS, vol. 11460, pp. 130–147. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_9
20. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Ann. Eugen. **7**(2), 179–188 (1936)
21. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_14
22. Goutte, C., Gaussier, E.: A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In: Losada, D.E., Fernández-Luna, J.M. (eds.) ECIR 2005. LNCS, vol. 3408, pp. 345–359. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31865-1_25
23. Hara, S., Hayashi, K.: Making tree ensembles interpretable. In: ICML Workshop on Human Interpretability in Machine Learning (WHI 2016) (2016)
24. Healy, A., Monahan, R., Power, J.F.: Evaluating the use of a general-purpose benchmark suite for domain-specific SMT-solving. In: Symposium on Applied Computing SAC 2016, pp. 1558–1561. ACM (2016). https://doi.org/10.1145/2851613.2851975
25. Jackson, D.: Alloy: a lightweight object modelling notation. Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)
26. Krings, S., Bendisposto, J., Leuschel, M.: From failure to proof: the ProB disprover for B and Event-B. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 199–214. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_15
27. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 361–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_23
28. Le Berre, D., Parrain, A.: The Sat4J library, release 2.2. J. Satisf. Boolean Model. Comput. **7**, 59–64 (2010). System description
29. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From animation to data validation: the ProB constraint solver 10 years on. In: Boulanger, J.-L. (ed.) Formal Methods Applied to Complex Systems: Implementation of the B Method, pp. 427–446. Wiley, Hoboken (2014)
30. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
31. Loh, W.: Classification and regression tree methods. In: Wiley StatsRef: Statistics Reference Online. American Cancer Society, September 2014. https://doi.org/10.1002/9781118445112.stat03886
32. Narayanan, I., et al.: SSD failures in datacenters: what? when? and why? In: Systems and Storage Conference, p. 7. ACM (2016)
33. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
34. Petrasch, J.: The decision does not fall far from the tree: automatic configuration of predicate solving. Master's thesis, Heinrich Heine Universität Düsseldorf, Universitätsstraße 1, 40225 Düsseldorf, April 2018
35. Plagge, D., Leuschel, M.: Validating B,Z and TLA$^+$ Using ProB and Kodkod. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 372–386. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_31

36. Schulz, S.: E-a brainiac theorem prover. Ai Commun. **15**(2,3), 111–126 (2002)
37. Strobl, C., Boulesteix, A.L., Zeileis, A., Hothorn, T.: Bias in random forest variable importance measures: Illustrations, sources and a solution. BMC Bioinf. **8**(1), 25 (2007). https://doi.org/10.1186/1471-2105-8-25
38. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
39. Yang, Q., Yin, J., Ling, C.X., Chen, T.: Postprocessing decision trees to extract actionable knowledge. In: International Conference on Data Mining, pp. 685–688. IEEE, November 2003. https://doi.org/10.1109/ICDM.2003.1251008