gym-autokey: RL for Rule-Based Deductive Program Verification in KeY

Andreas Boltres

Karlsruhe Institute of Technology
`https://www.kit.edu`

Introduction

Formal Verification

Formal Verification (FV) is the systematic process of proving or disproving the correctness of an artifact such as an algorithm or a software specification, using formal methods. These formal methods used are mathematically founded, e.g. logic calculi or deduction techniques.

FV applied to the specification of software/code is called *Program-* or *Software Verification* (SV). If the specification of a piece of code can be formally verified, it is guaranteed that the code's behaviour conforms to its specification, which can be a crucial quality aspect regarding performance, security and safety.

Automated Verification in KeY

The KeY Prover [1] is a deductive verification tool capable of verifying programs written in Java. Provided that the program's logic was formally specified and annotated using the Java Modeling Language (JML), deduction rules are applied to open formulas until either all have been proved or disproved.

The calculus of choice for the logical reasoning is the *sequent calculus*, which operates on sequences of the form $\phi_1, ..., \phi_n \Rightarrow \psi_1, ..., \psi_m$ (see [1], chapter 2.2.2 "Calculus"). The ϕ_i, ψ_i, called *formulas*, are given in first-order logic modulo theories. The theories found in the formulas model different parts of the Java language, such as integer arithmetic or strings (see [1], especially chapter 5 "Theories"). A sequent $\phi_1, ..., \phi_n \Rightarrow \psi_1, ..., \psi_m$ is valid iff the formula $\bigvee_{i=1}^{n} \phi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$ is valid.

KeY provides an automatic strategy that handles annotated programs in the following way: First, a program block is transformed into one or more formulas, using symbolic execution of program code included in the JML specification and an intermediate step in which the program and its specification are given in *dynamic logic for Java* (see [1], chapter 3). After the initial preparation, deduction rules are applied until all formulas have been reduced to a trivial form.

The automatic strategy of KeY provides a very accessible approach to formal verification of Java specifications, but for many specifications it can't fully end the proof because it doesn't have enough computational resources or not enough time. In these cases, the proof has to be resumed manually, which greatly reduces the ease of use for non-experts in formal methods.

Reinforcement Learning for Rule-Based Program Verification

Rule-Based program verification systems are diverse and inherently complex due to the nature of their problem domain. Therefore, the approach of applying RL techniques to such verification systems is made most versatile by executing all actions aside the actual rule applications outside the verification system.

Figure 1 shows the proposed structure for deep RL applications to this kind of verification system: Here, the only two tasks of the verification system are to apply the deduction steps given by the agent and to emit the open sequents (called *goals*). They are collected by a component that uses the new information about the goals to calculate the reward for the past deduction step and to extract the current state of the verification system.

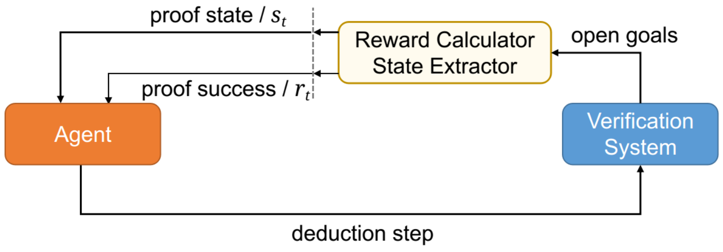

Fig. 1. An overview of the interplay between a rule-based program verifier and a deep RL system.

Non-Linearity / Episodes and Goals

It is not trivial to map the RL episode term to a process in the area of deductive rule-based verification. That is because the action flow of a single episode is inherently non-linear, but rather tree-like as there exist rule applications that may split a formula into multiple ones (see fig. 2).

In deductive program verification a specification results in multiple *proof obligations* (POs) which have to be proved in order to verify the specification. Every PO in turn consists of one or more sequents which have to be closed. The sequents are denoted as *goals*, and the initially open goals are denoted as *topgoals*. In fig. 2, the goals are identified by a unique number.

Goals that are split as a result of a deduction step become *parents*, and the resulting formulas become its *children*. If a goal has become a parent, it is considered successfully closed only if all children have been successfully closed;

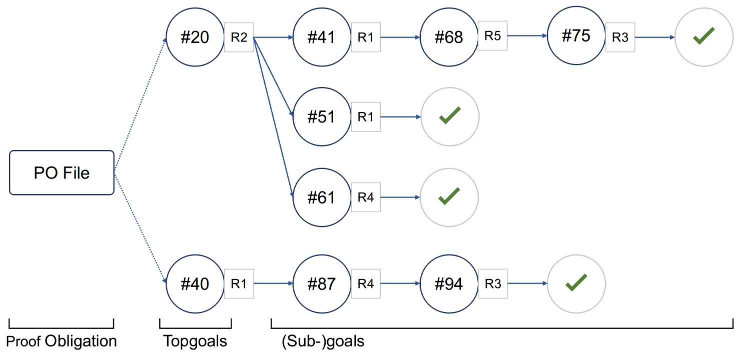

Fig. 2. An example episode of deductive rule-based verification. R1-R5 denote applications of different rules.

otherwise it is considered failed. Furthermore, all goals that are not topgoals stem from previously handled goals and are therefore denoted as *subgoals*.

Episode Bounds

First-order logic (the material our formulas are made of) in general is not decidable. This means that it is not possible to guarantee that every PO can be proved in a finite number of steps. For practical reasons, we hence assume that every PO that can be proved at all can be proved in a bounded and computationally feasible number of deduction steps.

An upper limit for deduction steps is imposed on each PO and the maximum depth of the proving tree of a topgoal is limited. We disregard every proof that exceeds these imposed limits, which means that we are learning the ability to practically prove POs instead of learning to determine the validity of them. As a consequence, verification episodes are finite.

If the limit of total deduction steps per PO is reached, the verification attempt of the whole PO is considered failed. Likewise, if the proof tree depth for a topgoal is reached, the proof attempt for the current topgoal is considered failed. If any goal proving attempt failed during an episode, e.g. though reaching these imposed proof size limits, the proof attempt of the whole PO is aborted and the episode ends.

Rewarding Scheme

A rewarding scheme for rule-based verification, in our opinion, should satisfy three aspects: It should encourage a successful closing of topgoals, it should discourage the failure to close topgoals and it should encourage to find shorter paths for goal closure. Moreover, it is the domain experts' opinion that splits should be encouraged to happen only if all resulting children can be closed. If any child of a split goal fails to be closed, the split is penalized.

In order to better handle splits and their effects on the proof status, we introduce the term *subepisode*: A subepisode is a segment of a proving tree starting from a topgoal or a goal directly resulting from a split, all containing all this goal's subgoals until closed or split again. In case of a split, each child marks the start of a new subepisode. In fig. 2, for example, the opening of goal #41 marks the start of a subepisode that ends with the closing of goal #75.

Name	Explanation
`open`	The newest goal is open and steps can still be made.
`success`	The newest goal (and thus the subepisode) has been closed successfully.
`fail`	The newest goal is open and steps cannot be made anymore (e.g. proof size/tree depth limit reached).
`crash`	The verification system crashed or timed out.
`parent`	The goal split into multiple subgoals.

Table 1. Statuses for subepisodes.

Using the notion of subepisodes, the status of an episode is determined through the subepisodes it creates along the way. In order to retrieve the status for steps that split the currently open goal of a subepisode `e`, the resulting child goals and their respective subepisodes are worked through before assigning the status to the splitting step in `e`, as all children need to terminate successfully to propagate their success to the parent.

Training and Test Data

In order to train effectively, the agent must be fed realistic data. For the domain of rule-based program verification, this would be formulas from the domain of first-order logic, stemming from a given set of specified programs. A set of 268 specification files has been prepared for a direct pick-up by the automatic mode of KeY, containing 1322 topgoals. Their file paths are put into different text files in the folder `data/po_files`, each containing a certain set of files.

Features

When presented with a logical formula, it is the authors opinion that the best verification step to be applied can be derived from just the semantics of the formula. A consideration of the state of the verifier or the rule application history thus is only necessary if the rule/step is chosen by trial-and-error or if there is human uncertainty about the effects of applications or the verifier's configuration. As a result, a learned agent controller should be able to make informed decisions with just a representation of the current formula.

Within the scope KeY, the formula of a goal is handled as an AST (see fig. 3) where a node represents a symbol like a function, a quantifier or a constant, and its children represent its parameters or the symbols bound to it. Constants, numbers and similar symbols are represented as leaf nodes, as they don't bind any symbols to themselves. The root node of the AST is the implication linking antecedent and succedent, which in turn are modeled by a junction of one or more formulas or spared out if containing no formulas.

$$(\forall x : f(f(x)) = f(x)) \Rightarrow f(f(f(c))) = f(c)$$

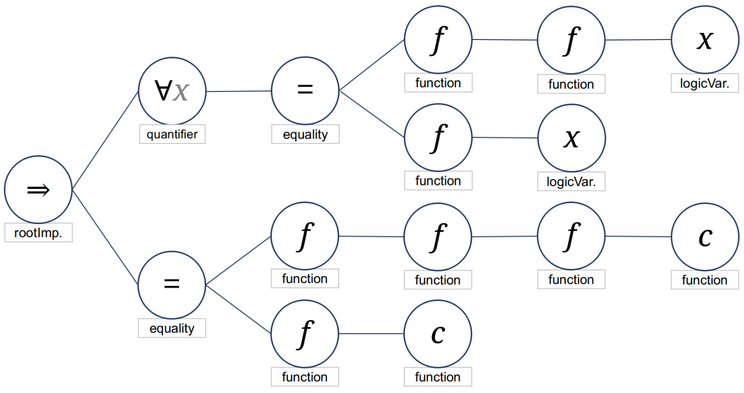

Fig. 3. An example formula and its AST.

When designing features for these formula AST, part of the semantics is extractable from the general shape of the AST and from the occurrences of certain symbols or symbol combinations. While domain knowledge is needed to determine which symbol categories are relevant and which are not, the features elicited for these categories are held as generic as possible, since the algorithm might discover regularities previously hidden from the domain experts. See table 2 for the categories considered and table 3 for the complete list of features that are extracted from the formula ASTs.

Name	Contains
equality	equality operator: `=`
inequality	inequality operators: `<`, `>`,`<=`,`>=`
heap	any symbol containing `heap` in its name
anonymity	any symbol containing `anon` in its name
numbers	the function with the name `Z`, which identifies numbers
programConstants	any symbol of the class *programConstants*
modalities	any symbol of the class *modalities*
ITE	the symbols `IfThenElse` and `IfExThenElse`
depContracts	any symbol of the classes *ProgramMethod* and *ObersverFunction*
quantifiers	the quantifiers ∀ and ∃

Table 2. The predetermined feature categories and which symbol occurrences are included in them.

Feature f	Explanation	μ_f	σ_f
astSize	The number of nodes of an AST.	314.483	324.699
astHeight	A relative AST height measure that uses the cumulated depth of all nodes: -1 indicates the widest possible tree, +1 the tallest possible tree.	-0.876	0.089
formulaCount	The number of formulas the antecedent and succedent hold.	16.925	12.137
selectStoreCnt	The number of occurrences of the combined `select(store())` function.	0.135	1.368
isPresent	+1 if symbols of a category are present in this AST, -1 otherwise.	-	-
balance	Smaller if the occurrences of a category are found more to the left side of the AST, bigger if found more to the right side. Ranges from -1 to +1.	-	-
spread	Smaller if occurrences of a category are found concentrated in a single spot, bigger if they are more spread out. Ranges from -1 to +1.	-	-
relCount	Number of category occurrences divided by total symbol count.	< 0.103	< 0.059
relDepth	Average depth of category occurrences divided by average depth of all symbols.	< 0.667	< 0.133

Table 3. The features elicited, their explanations and distribution information for non-normalized features. The upper half contains the general AST features while the lower half contains the per-category features.

All in all, a total of 54 features is observed. Because they differ in their definition range and characteristic values, a normalization process is undertaken for every feature but the *present*, the *spread* and the *balance* feature: Using each feature f's values x_f for all topgoals in the available data set, the feature's mean μ_f and standard deviation σ_f are computed. When extracting features for the learning or usage process afterwards, these figures are fed into a feature-specific tanh function:

$$x'_f = tanh(\frac{x_f - \mu_f}{2\sigma_f}) \tag{1}$$

As a result, the normalized values x'_f of every feature f range from -1 to 1, which might optimize the numerical stability of underlying deep learning policies.

Actions: Tactics

In KeY, the whole set of applicable rules consists of over 1000 different deduction rules. Hence, for the sake of simplicity, we introduced the concept of *tactics* to KeY. A tactic is a collection of deduction rules for simplification and reshaping, also including a procedure description on how to apply this set of rules on a formula. This means that the selection of a tactic also applies the included rules according to given description. Tactics are manually created, each covering a certain area or theory of the underlying formula logic or behaving in a distinctive way (see table 4). The tactics ***INT***, ***HEAP*** and ***EQUALITY*** are specialized subsets of ***AUTO***, which means that ***AUTO*** can do everything the other three tactics can. Albeit, it works at a considerably slower pace.

Name	Explanation
AUTO	Mimics the automatic strategy currently used by KeY.
AUTO_NOSPLIT	Same as ***AUTO*** but avoids goal splits.
SMT	Outsources the current goal to an SMT solver.
MODELSEARCH	Tries to find variable instantiations that satisfy the goal.
DEPENDENCY	Tries to resolve program dependencies present in the goal.
EXPAND	Expands a function definition.
QUANT	Instantiates quantors found in the goal.
HEAP	Subset of ***AUTO***'s rules, focusing on memory allocation and usage statements.
INT	Subset of ***AUTO***'s rules, focusing on integer arithmetic.
EQUALITY	Subset of ***AUTO***'s rules, focusing on equalities.

Table 4. Tactics implemented in KeY.

The verification process using tactics is analogous to using singular rules, in that selecting a tactic can produce new goals and depth-first search is employed until no more goals are open. Likewise, tactic selections which have no effect leave the currently active goal as-is.

Related Work

ML for Methods of Formal Verification

Due to the complexity of the problem domain, the application of ML to FV has only picked up pace in the last few years, although first attempts have already been made decades ago [25].

In the survey paper of Amrani et al. (2018), a general insight is given into the impact of ML techniques when applied to various fields within the FV domain [2]. According to the authors, areas in which ML has brought improvements include SAT/SMT solving (e.g. [29], [20] and [35]), Theorem Proving (e.g [6], [23] and [24]), Model Checking (e.g. [3], [9]) and Static Analysis (e.g. [36],[22]).

The area of Software Verification is most similar to Amrani's area of Theorem Proving, dealing with the (dis-)proving of software/system semantics which have been expressed in mathematical theories. For some SV tools (included in the comprehensive participants list of the **VerifyThis** challenge [12]) there exist augmentation approaches which use ML:

For the Isabelle proof assistant [34], Blanchette et al. have presented a new theorem selector that uses the Sparse Naive Bayes and k-Nearest Neighbors algorithms to select a suitable subset of all usable theorems. [5].

For Why3 [13], a program verification tool outsourcing the proving of theorems to a variety of automated Theorem Provers, the Where4 extension estimates runtime figures using regression and chooses the ATP with the best runtime prediction [18].

For the higher-order logic prover Satallax [8], Färber and Brown introduced a proof guidance method adjusting the delay with which propositions are processed, based on the classification result of a naive-bayes classifier [15].

Deep Reinforcement Learning

RL, especially with the fusion of deep learning methods, has been credited for a large part of the recent advances in Artificial Intelligence. Its application improved the state of the art in domains such as generating text ([16]), playing (video) games ([28], [32]) or navigating robots ([21]).

Perhaps most similar to our domain is the work of Crouse et al. [10]. There, RL is used for internal guidance in a rule-based reasoning process: the agent decides at each step which inference rule to apply to a clause.

The results indicate that the performance of such a neural network trained with RL is comparable to existing reasoners, although slightly worse than the reasoner is was compared with (which is Beagle [4]).

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Amrani, M., Lúcio, L., Bibal, A.: ML + FV = ♡? A Survey on the Application of Machine Learning to Formal Verification. arXiv:1806.03600 [cs] (Jun 2018)
3. Araragi, T., Cho, S.M.: Checking Liveness Properties of Concurrent Systems by Reinforcement Learning. In: Edelkamp, S., Lomuscio, A. (eds.) Model Checking and Artificial Intelligence, 4th Workshop, MoChArt IV, Riva Del Garda, Italy, August 29, 2006, Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 4428, pp. 84–94. Springer (2006). https://doi.org/10.1007/978-3-540-74128-2_6
4. Baumgartner, P., Bax, J., Waldmann, U.: Beagle – A Hierarchic Superposition Theorem Prover. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25. pp. 367–377. Lecture Notes in Computer Science, Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_25
5. Blanchette, J.C., Greenaway, D., Kaliszyk, C., Kühlwein, D., Urban, J.: A Learning-Based Fact Selector for Isabelle/HOL. J Autom Reasoning **57**(3), 219–244 (Oct 2016). https://doi.org/10.1007/s10817-016-9362-8

6. Bridge, J.P., Holden, S.B., Paulson, L.C.: Machine Learning for First-Order Theorem Proving: Learning to Select a Good Heuristic. J Autom Reasoning **53**(2), 141–172 (Aug 2014). https://doi.org/10.1007/s10817-014-9301-5
7. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. arXiv preprint arXiv:1606.01540 (2016)
8. Brown, C.E.: Satallax: An Automatic Higher-Order Prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning. pp. 111–117. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_11
9. Clarke, E., Gupta, A., Kukula, J., Strichman, O.: SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification. pp. 265–279. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2002)
10. Crouse, M., Abdelaziz, I., Makni, B., Whitehead, S., Cornelio, C., Kapanipathi, P., Pell, E., Srinivas, K., Thost, V., Witbrock, M., Fokoue, A.: A Deep Reinforcement Learning based Approach to Learning Transferable Proof Guidance Strategies. arXiv:1911.02065 [cs] (Feb 2020), `http://arxiv.org/abs/1911.02065`, arXiv: 1911.02065
11. Dulov, O.: bwcloud: cross-site server virtualization (2016)
12. Ernst, G., Huisman, M., Mostowski, W., Ulbrich, M.: VerifyThis – Verification Competition with a Human Factor. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 176–195. Lecture Notes in Computer Science, Springer International Publishing (2019)
13. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 125–128. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2013)
14. Francois-Lavet, V., Henderson, P., Islam, R., Bellemare, M.G., Pineau, J.: An Introduction to Deep Reinforcement Learning. FNT in Machine Learning **11**(3-4), 219–354 (2018). https://doi.org/10.1561/2200000071, `http://arxiv.org/abs/1811.12560`, arXiv: 1811.12560
15. Färber, M., Brown, C.: Internal Guidance for Satallax. arXiv:1605.09293 [cs] (May 2016), `http://arxiv.org/abs/1605.09293`, arXiv: 1605.09293
16. Guo, H.: Generating Text with Deep Reinforcement Learning. arXiv:1510.09202 [cs] (Oct 2015), `http://arxiv.org/abs/1510.09202`, arXiv: 1510.09202
17. van Hasselt, H., Guez, A., Silver, D.: Deep Reinforcement Learning with Double Q-learning. arXiv:1509.06461 [cs] (Dec 2015), `http://arxiv.org/abs/1509.06461`, arXiv: 1509.06461
18. Healy, A., Monahan, R., Power, J.F.: Predicting SMT Solver Performance for Software Verification. Electron. Proc. Theor. Comput. Sci. **240**, 20–37 (Jan 2017). https://doi.org/10.4204/EPTCS.240.2
19. Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y.: Stable baselines. `https://github.com/hill-a/stable-baselines` (2018)
20. Hutter, F., Hamadi, Y., Hoos, H.H., Leyton-Brown, K.: Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming - CP 2006. pp. 213–228. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2006)

21. Kahn, G., Villaflor, A., Ding, B., Abbeel, P., Levine, S.: Self-Supervised Deep Reinforcement Learning with Generalized Computation Graphs for Robot Navigation. In: 2018 IEEE International Conference on Robotics and Automation (ICRA). pp. 5129–5136 (May 2018). https://doi.org/10.1109/ICRA.2018.8460655, iSSN: 2577-087X
22. Kim, S., Jr, E.J.W., Zhang, Y.: Classifying Software Changes: Clean or Buggy? IEEE Transactions on Software Engineering **34**(2), 181–196 (Mar 2008). https://doi.org/10.1109/TSE.2007.70773
23. Kühlwein, D., Urban, J.: MaLeS: A Framework for Automatic Tuning of Automated Theorem Provers. J Autom Reasoning **55**(2), 91–116 (Aug 2015). https://doi.org/10.1007/s10817-015-9329-1
24. Kühlwein, D., van Laarhoven, T., Tsivtsivadze, E., Urban, J., Heskes, T.: Overview and Evaluation of Premise Selection Techniques for Large Theory Mathematics. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning. pp. 378–392. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2012)
25. Lounis, H.: Integrating machine-learning techniques in knowledge-based systems verification. In: Komorowski, J., Raś, Z.W. (eds.) Methodologies for Intelligent Systems. pp. 405–414. Lecture Notes in Computer Science, Springer Berlin Heidelberg (1993)
26. Mitchell, T.M.: Machine Learning. McGraw-Hill Series in Computer Science, McGraw-Hill, New York (1997)
27. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs] (Dec 2013), `http://arxiv.org/abs/1312.5602`, arXiv: 1312.5602
28. Oh, J., Guo, X., Lee, H., Lewis, R.L., Singh, S.: Action-Conditional Video Prediction using Deep Networks in Atari Games. In: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (eds.) Advances in Neural Information Processing Systems 28, pp. 2863–2871. Curran Associates, Inc. (2015)
29. Samulowitz, H., Memisevic, R.: Learning to Solve QBF. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada. pp. 255–260. AAAI Press (2007)
30. Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized Experience Replay. arXiv:1511.05952 [cs] (Feb 2016), `http://arxiv.org/abs/1511.05952`, arXiv: 1511.05952
31. Schmidhuber, J.: Deep Learning in Neural Networks: An Overview. Neural Networks **61**, 85–117 (Jan 2015). https://doi.org/10.1016/j.neunet.2014.09.003, `http://arxiv.org/abs/1404.7828`, arXiv: 1404.7828
32. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G.v.d., Graepel, T., Hassabis, D.: Mastering the game of Go without human knowledge. Nature **550**(7676), 354–359 (Oct 2017). https://doi.org/10.1038/nature24270, `https://www.nature.com/articles/nature24270`, number: 7676 Publisher: Nature Publishing Group
33. Sutton, R.S., Barto, A.G.: Reinforcement Learning - an Introduction. Adaptive Computation and Machine Learning, MIT Press (1998)
34. Technische Universität München: Isabelle. https://isabelle.in.tum.de/
35. Wu, H.: Improving SAT-solving with Machine Learning. In: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education. pp. 787–788. SIGCSE '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3017680.3022464

36. Yerima, S.Y., Sezer, S., Muttik, I.: Android Malware Detection Using Parallel Machine Learning Classifiers. 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies pp. 37–42 (Sep 2014). https://doi.org/10.1109/NGMAST.2014.23