Goéland: a Concurrent Tableau-Based Theorem Prover (System Description)

Julie Cailler, Johann Rosain, David Delahaye, Simon Robillard and Hinde Bouziane

August 12, 2022

Goéland : A Concurrent Tableau-Based Theorem Prover (System Description)

Julie Cailler[1][0000−0002−6665−8089], Johann Rosain[1][0000−0003−1719−2654], David Delahaye[1][0000−0003−4779−1359], Simon Robillard[1][0000−0003−4751−380X], and Hinde Lilia Bouziane[1][0000−0002−8749−4562]

LIRMM, Univ Montpellier, CNRS, Montpellier, France
Firstname.Lastname@lirmm.fr

Abstract. We describe Goéland, an automated theorem prover for first-order logic that relies on a concurrent search procedure to find tableau proofs, with concurrent processes corresponding to individual branches of the tableau. Since branch closure may require instantiating free variables shared across branches, processes communicate via channels to exchange information about substitutions used for closure. We present the proof search procedure and its implementation, as well as experimental results obtained on problems from the TPTP library.

Keywords: Automated Theorem Proving · Tableaux · Concurrency.

1 Introduction

Although clausal proof techniques have enjoyed success in automated theorem proving, some applications benefit from reasoning on unaltered formulas (rather than Skolemized clauses), while others require the production of proofs in a sequent calculus. These roles are fulfilled by provers based on the tableau method [17], as initially designed by Beth and Hintikka [2, 13]. For first-order logic, efficient handling of universal formulas is typically achieved with free variables that are instantiated only when needed to close a branch. This step is said to be *destructive* because it may affect open branches sharing variables. This causes fairness (and consequently, completeness) issues, as illustrated in Figure 1. In this example, exploring the left branch produces a substitution that prevents direct closure of the right branch. Reintroducing the original quantified formula with a different free variable is not sufficient to close the right branch, because an applicable δ-rule creates a new Skolem symbol that will result in a different but equally problematic substitution every time a left branch is explored. Thus, systematically exploring the left branch before the right leads to non-termination of the search. Conversely, exploring the right branch first produces a substitution (which instantiates the free variable X with a rather than b) that closes both branches.

Concurrent computing offers a way to implement a proof search procedure that explores branches simultaneously. Such a procedure can compare closing

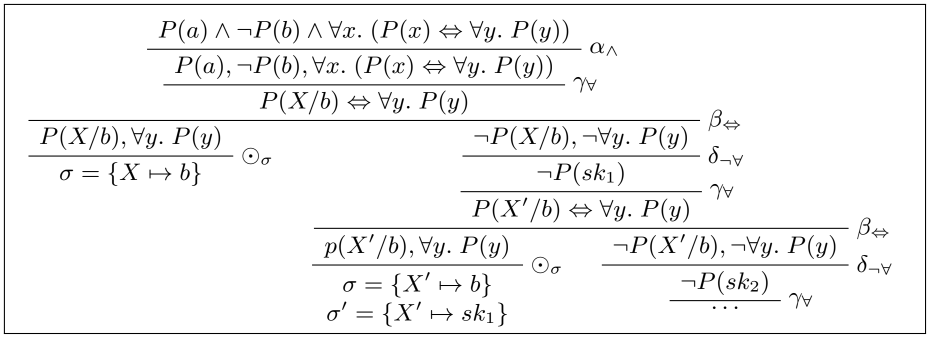

Fig. 1. Incompleteness caused by unfair selection of branches

substitutions to detect (dis)agreements between branches, and consequently either close branches early, or restart proof attempts with limited backtracking. The simultaneous exploration of branches is handled by the concurrency system, either by interleaving computations through scheduling, or by executing tasks in parallel if the hardware resources allow it. A concurrent procedure naturally lends itself to parallel execution, allowing us to take advantage of multi-core architectures for efficient first-order theorem proving. Thus, concurrency provides an elegant and efficient solution to proof search with free variable tableaux.

In this paper, we describe a concurrent destructive proof search procedure for first-order analytic tableaux (Section 2) and its implementation in a tool called Goéland, as well as its evaluation on problems from the TPTP library [19] and comparison to other state-of-the-art provers (Section 3).

Related Work A lot of research has been carried out on the parallelization of proof search procedures [4], often focusing primarily on parallel execution and performance. In contrast, we use concurrency not only as a way to take advantage of multi-core architectures, but also as an algorithmic device that is useful even for sequential execution (with interleaved threads). Some concurrent and parallel approaches focus more distinctly on the exploration of the search space, either by dividing the search space between processes (*distributed search*) or by using processes with different search plans on the same space (*multi search*) [3]. These approaches can be performed either by *heterogeneous systems* that rely on cooperation between systems with different inference systems [1,8,12], or *homogeneous systems* where all deductive processes use the same inference system. According to this classification, the technique presented here is a homogeneous system that performs a distributed search. Concurrent tableaux provers include the model-elimination provers CPTheo [12] and Partheo [18], and the higher-order prover Hot [15], which notably uses concurrency to deal with fairness issues arising from the non-terminating nature of higher-order unification. Lastly, concurrency has been used as the basis of a generic framework to present various proof strategies [10] or allow distributed calculations over a network [21].

2 Concurrent Proof Search

Free Variable Tableaux Goéland attempts to build a refutation proof for a first-order formula, i.e., a closed tableau for its negation, using a standard free-variable tableau calculus [11]. The calculus is composed of α-, γ- and δ-rules that extend a branch with one formula, β-rules that divide a branch by extending it with two formulas, and a $\odot$-rule that closes a branch. γ-rules deal with universally-quantified formulas by introducing a formula with a free variable. A free variable is not universally quantified, but is instead a placeholder for some term instantiation, typically determined upon branch closure. δ-rules deal with existentially-quantified formulas by introducing a formula with a Skolem function symbol that takes as arguments the free variables in the branch. This ensures freshness of the Skolem symbol independently of variable instantiation.

The branch closure rule applies to a branch carrying atomic formulas P and Q such that, for some substitution σ, $\sigma(P) = \sigma(\neg Q)$. In that case, σ is applied to all branches. That rule is consequently *destructive*: applying a substitution to close one branch may modify another, removing the possibility to close it immediately. A tableau is closed when all its branches are closed. Closing a tableau can thus be seen as providing a global unifier that closes all branches.

Semantics for Concurrency Goéland relies on a concurrent search procedure. In order to present this procedure, we use a simple WHILE language augmented with instructions for concurrency, in the style of CSP [14]. Each process has its own variable store, as well as a collection of process identifiers used for communication: π_{parent} denotes the identifier of a process's parent, while Π_{children} denotes the collection of identifiers of active children of that process. Given a process identifier π and an expression e, the command $\pi\,!\,e$ is used to send an asynchronous message with the value e to the process identified by π. Conversely, the command $\pi\,?\,x$ blocks the execution until the process identified by π sends a message, which is stored in the variable x. Lastly, the instruction **start** creates a new process that executes a function with some given arguments, while the instruction **kill** interrupts the execution of a process according to its identifier.

Proof Search Procedure The proof search is carried out concurrently by processes corresponding to branches of the tableau. Processes are started upon application of a β-rule, one for each new branch. Communications between processes take two forms: a process may send a set of closing substitutions for its branch to its parent, or a parent may send a substitution (that closes one of its children's branch) to the other children. The proof search is performed by the *proofSearch*, *waitForParent*, and *waitForChildren* procedures (described in Procedures 1, 2, and 3, respectively).

The *proofSearch* procedure initiates the proof search for a branch. It first attempts to apply the closure rule. A closing substitution is called *local* to a process if its domain includes only free variables introduced by this process or one of its descendants (i.e., if the variables do not occur higher in the proof tree). If one of the closing substitutions is local to the process, it is reported and

Procedure 1: *proofSearch*

```
   Data: a tableau $T$
 1 begin
 2     var $\Theta \leftarrow applyClosingRule(T)$ ;
 3     for $\theta \in \Theta$ do
 4         if $isLocal(\theta)$ then
 5             $\pi_{\text{parent}}$ ! $\{\theta\}$
 6             return
 7     if $\Theta \neq \emptyset$ then
 8         $\pi_{\text{parent}}$ ! $\Theta$
 9         $waitForParent(T, \Theta)$
10     else if $applicableAlphaRule(T)$ then
11         $proofSearch(applyAlphaRule(T))$
12     else if $applicableDeltaRule(T)$ then
13         $proofSearch(applyDeltaRule(T))$
14     else if $applicableBetaRule(T)$ then
15         for $T' \in applyBetaRule(T)$ do
16             start $proofSearch(T')$
17         $waitForChildren(T, \emptyset, \emptyset)$
18     else if $applicableGammaRule(T)$ then
19         $proofSearch(applyGammaRule(T))$
20     else
21         $\pi_{\text{parent}}$ ! $\emptyset$
```

the process terminates. If only non-local closing substitutions are found, they are reported and the process executes *waitForParent*. Otherwise, the procedure applies tableau expansion rules according to the priority: $\alpha \prec \delta \prec \beta \prec \gamma$. If a β-rule is applied, new processes are started, and each of them executes *proofSearch* on the newly created branch, while the current process executes *waitForChildren*.

The *waitForParent* procedure is executed by a process after it has found closing non-local substitutions. Such substitutions may prevent closure in other branches. In these cases, the parent will eventually send another candidate substitution. *waitForParent* waits until such a substitution is received, and triggers a new step of proof search. The process may also be terminated by its parent (via the **kill** instruction) during the execution of this procedure, if one of the substitutions previously sent by the process leads to closing the parent's branch.

The *waitForChildren* procedure is executed by a process after the application of a β-rule and the creation of child processes. The set of substitutions sent by each child is stored in a map *subst* (Line 2), initially undefined everywhere ($\mathrm{f}_\perp$). This procedure closes the branch (Line 13) if there exists a substitution θ that agrees with one closing substitution of each child process, i.e., for each child process, the process has reported a substitution σ such that $\sigma(X) = \theta(X)$ for any variable X in the domain of σ. If no such substitution can be found after all the children have closed their branches, then one closing substitution

Procedure 2: *waitForParent*

```
Data: a tableau $T$, a set $\Theta_{\text{sent}}$ of substitutions sent by this process to its parent
1 begin
2     $\pi_{\text{parent}}$ ? $\sigma$
3     if $\sigma \in \Theta_{sent}$ then
4         $\pi_{\text{parent}}$ ! $\sigma$
5         $waitForParent(T, \Theta_{\text{sent}})$
6     else
7         $proofSearch(\sigma(T))$
```

$\sigma \in subst$ is picked arbitrarily (Line 18) and sent to all the children (which are at that point executing *waitForParent*) to restart their proof attempts. With the additional constraint of the substitution σ, the new proof attempts may fail, hence the necessity for backtracking among candidate substitutions $\Theta_{\text{backtrack}}$ (Line 5 and 6). At the end, if all the substitutions were tried and failed, the process sends a failure message (symbolized by $\emptyset$) to its parent.

Thus, concurrency and backtracking are used to prevent incompleteness resulting from unfair instantiation of free variables. Another potential source of unfairness is the γ-rule, when applied more than once to a universal formula (reintroduction). This may be needed to find a refutation, but unbounded reintroductions would lead to unfairness. Iterative deepening [16] is used to guard against this: a bound limits the number of reintroductions on any single branch, and if no proof is found, the bound is increased and the proof search restarted.

Figure 2 illustrates the interactions between processes for the problem in Figure 1, and shows how concurrency helps ensure fairness. It describes the parent process, in the top box, and below, the two children processes created upon application of the β-rule. Dotted lines separate successive states of a process (i.e., Procedures 1, 2 and 3 seen above), while arrows and boxes represent substitution exchanges. The number above each arrow indicates the chronology of the interaction. After both children have returned a substitution (1), the parent arbitrarily chooses one of them, starting with $X \mapsto b$, and sends it to the children (2). Since this substitution prevents closure in the right branch (3), the parents later backtracks and sends the other substitution $X \mapsto a$ (4), allowing both children (5) and then the parent to close successfully.

3 Implementation and Experimental Results

Implementation The procedures presented in Section 2 are implemented in the Goéland prover[1] using the Go language. Go supports concurrency and parallelism, based on lightweight execution threads called *goroutines* [20]. Goroutines are executed according to a so-called *hybrid threading* (or $M : N$) model: M

[1] Available at: https://github.com/GoelandProver/Goeland/releases/tag/v1.0.0-beta.

Procedure 3: *waitForChildren*

```
Data: a tableau $T$, a set $\Theta_{\text{sent}}$ of substitutions sent by this process to its
      parent, a set $\Theta_{\text{backtrack}}$ of substitutions used for backtracking
begin
    var subst $\leftarrow$ $\text{f}_\perp$
    while $\exists \pi \in \mathit{\Pi}_{children}.\ \text{subst}[\pi] = \perp$ do
        $\pi$ ? subst[$\pi$]
        if subst[$\pi$] $= \emptyset$ then
            if $\exists \theta \in \Theta_{backtrack}$ then
                for $\pi \in \mathit{\Pi}_{children}$ do $\pi$ ! $\theta$;
                $\mathit{waitForChildren}(T, \Theta_{\text{sent}}, \Theta_{\text{backtrack}} \setminus \{\theta\})$
            else
                for $\pi \in \mathit{\Pi}_{children}$ do kill $\pi$;
                $\pi_{\text{parent}}$ ! $\emptyset$
                return

    if $\exists \theta, \text{agreement}(\theta, \text{subst})$ then
        $\pi_{\text{parent}}$ ! $\{\theta\}$
        for $\pi \in \mathit{\Pi}_{children}$ do kill $\pi$;
        $\mathit{waitForParent}(T, \Theta_{\text{sent}} \cup \{\theta\})$
    else
        $\sigma \leftarrow \text{choice(subst)}$
        for $\pi \in \mathit{\Pi}_{children}$ do $\pi$ ! $\sigma$;
        $\mathit{waitForChildren}(T, \Theta_{\text{sent}}, \Theta_{\text{backtrack}} \cup \bigcup_{\pi} \text{subst}[\pi] \setminus \{\sigma\}))$
```

goroutines are executed over N effective threads and scheduling is managed by both the Go runtime and the operating system. This threading model allows the execution of a large number of goroutines with a reasonable consumption of system resources. Goroutines use channels to exchange messages, so that the implementation is close to the presentation of Section 2.

Goéland has, for the time being, no dedicated mechanism for equality reasoning. However, we have implemented an extension that implements deduction modulo theory [9], i.e., transforms axioms into rewrite rules over propositions and terms. Deduction modulo theory (DMT) has proved very useful to improve proof search when integrated into usual automated proof techniques [5], and also produces excellent results with manually-defined rewrite rules [6, 7]. In Goéland, deduction modulo theory selects some axioms on the basis of a simple syntactic criterion and replaces them by rewrite rules.

Experimental Results We evaluated Goéland on two problems categories with FOF theorems in the TPTP library (v7.4.0): syntactic problems without equality (SYN) and problems of set theory (SET). The former was chosen for its elementary nature, whereas the latter was picked primarily to evaluate the performance of the deduction modulo theory, as the axioms of set theory are good targets for rewriting. We compared the results with those of five other provers: tableau-

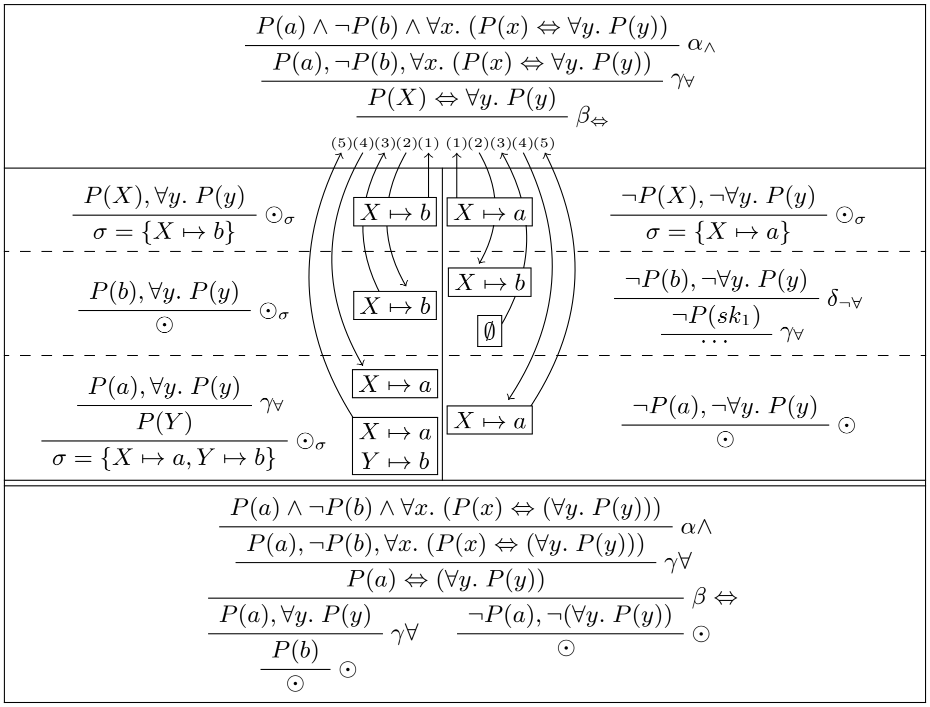
$P(a) \land \neg P(b) \land \forall x.\ (P(x) \Leftrightarrow \forall y.\ P(y))$
$\alpha_\land$
$P(a), \neg P(b), \forall x.\ (P(x) \Leftrightarrow \forall y.\ P(y))$
$\gamma_\forall$
$P(X) \Leftrightarrow \forall y.\ P(y)$
$\beta_\Leftrightarrow$
(5)(4)(3)(2)(1)
(1)(2)(3)(4)(5)
$P(X), \forall y.\ P(y)$
$\sigma = \{X \mapsto b\}$
$\odot_\sigma$
$X \mapsto b$
$X \mapsto a$
$\neg P(X), \neg\forall y.\ P(y)$
$\sigma = \{X \mapsto a\}$
$\odot_\sigma$
$P(b), \forall y.\ P(y)$
$\odot$
$\odot_\sigma$
$X \mapsto b$
$X \mapsto b$
$\emptyset$
$\neg P(b), \neg\forall y.\ P(y)$
$\neg P(sk_1)$
$\delta_{\neg\forall}$
$\cdots$
$\gamma_\forall$
$P(a), \forall y.\ P(y)$
$P(Y)$
$\gamma_\forall$
$\sigma = \{X \mapsto a, Y \mapsto b\}$
$\odot_\sigma$
$X \mapsto a$
$X \mapsto a$
$Y \mapsto b$
$X \mapsto a$
$\neg P(a), \neg\forall y.\ P(y)$
$\odot$
$\odot$
$P(a) \land \neg P(b) \land \forall x.\ (P(x) \Leftrightarrow (\forall y.\ P(y)))$
$\alpha\land$
$P(a), \neg P(b), \forall x.\ (P(x) \Leftrightarrow (\forall y.\ P(y)))$
$\gamma\forall$
$P(a) \Leftrightarrow (\forall y.\ P(y))$
$\beta \Leftrightarrow$
$P(a), \forall y.\ P(y)$
$\gamma\forall$
$P(b)$
$\odot$
$\odot$
$\neg P(a), \neg(\forall y.\ P(y))$
$\odot$
$\odot$

Fig. 2. Proof search and resulting proof for $P(a) \land \neg P(b) \land \forall x.(P(x) \Leftrightarrow \forall y.P(y))$

based provers Zenon (v0.8.5), Princess (v2021-05-10), as well as saturation-based provers LeoIII (v1.6), E (v2.6) and Vampire (v4.6.1). Experiments were executed on a computer equipped with an Intel Xeon E5-2680 v4 2.4GHz 2×14-core processor and 128 GB of memory. Each proof attempt was limited to 300 seconds. Table 1 and Figure 3 report the results. Table 1 shows the number of problems solved by each prover, the cumulative time, and the number of problems solved by a given prover but not by Goéland (+) and conversely (−). Figure 3 presents the cumulative time required to solve the number of problems.

As can be observed, the results of Goéland are comparable to, or slightly better than those of other tableau-based provers on problems from SYN, while saturation theorem provers achieve the best results. On this category, the axioms do not trigger deduction modulo theory rewriting rules, hence the similar results of Goéland and Goéland+DMT. On SET, Goéland+DMT obtains significantly better results than other tableau-based provers. This confirms the previous results on the performance of deduction modulo theory for set theory [6, 7].

4 Conclusion

We have presented a concurrent proof search procedure for tableaux in first-order logic with the aim of ensuring a fair exploration of the search space. This

	SYN (263 problems)		SET (464 problems)	
Goéland	**199** (190 s)		**150** (4659 s)	
Goéland+DMT	**199** (196 s)	(+0, −0)	**278** (1292 s)	(+142, −14)
Zenon	**256** (67 s)	(+60, −3)	**150** (562 s)	(+75, −75)
Princess	**195** (189 s)	(+1, −5)	**258** (1168 s)	(+141, −33)
LeoIII	**195** (268 s)	(+1, −5)	**177** (2925 s)	(+77, −50)
E	**261** (168 s)	(+62, −0)	**363** (2377 s)	(+223, −10)
Vampire	**262** (13 s)	(+63, −0)	**321** (4122 s)	(+188, −17)

Table 1. Experimental results over the TPTP library

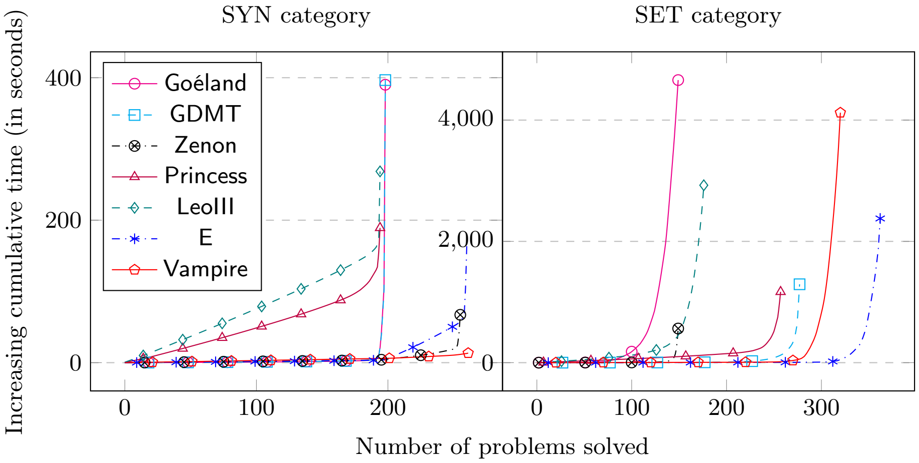

Fig. 3. Cumulative time per problem solved between Goéland, Goéland+DMT(GDMT), Zenon, Princess, LeoIII, E, and Vampire

procedure has been implemented in the prover Goéland. This tool is still in an early stage, and (with the exception of deduction modulo theory) implements only the most basic functionalities, yet empirical results are encouraging. We plan on adding functionalities such as equality reasoning, arithmetic reasoning, and support for polymorphism to Goéland, which should increase its usability and performance. The integration of these functionalities in the context of a concurrent prover seems to be a promising line of research. Further investigation is also needed to prove the fairness, and therefore completeness, of our procedure.

References

1. Benzmüller, C., Kerber, M., Jamnik, M., Sorge, V.: Experiments with an Agent-Oriented Reasoning System. In: Annual Conference on Artificial Intelligence. Lecture Notes in Computer Science (LNCS), vol. 2174, pp. 409–424. Springer (2001)

2. Beth, E.W.: Formal Methods: An Introduction to Symbolic Logic and to the Study of Effective Operations in Arithmetic and Logic, Synthese Library, vol. 4. D. Reidel Pub. Co. (1962)
3. Bonacina, M.P.: A Taxonomy of Parallel Strategies for Deduction. Annals of Mathematics and Artificial Intelligence **29**(1), 223–257 (2000)
4. Bonacina, M.P.: Parallel Theorem Proving. In: Handbook of Parallel Constraint Reasoning, pp. 179–235. Springer (2018)
5. Burel, G., Bury, G., Cauderlier, R., Delahaye, D., Halmagrand, P., Hermant, O.: First-Order Automated Reasoning with Theories: When Deduction Modulo Theory Meets Practice. Journal of Automated Reasoning (JAR) **64**(6), 1001–1050 (2020)
6. Bury, G., Cruanes, S., Delahaye, D., Euvrard, P.L.: An Automation-Friendly Set Theory for the B Method. In: Abstract State Machines, Alloy, B, VDM, and Z (ABZ). Lecture Notes in Computer Science (LNCS), vol. 10817, pp. 409–414. Springer (2018)
7. Bury, G., Delahaye, D., Doligez, D., Halmagrand, P., Hermant, O.: Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence and Reasoning (LPAR). EPiC Series in Computing, vol. 35, pp. 42–58. EasyChair (2015)
8. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT – A Distributed and Learning Equational Prover. Journal of Automated Reasoning **18**(2), 189–198 (1997)
9. Dowek, G., Hardin, T., Kirchner, C.: Theorem Proving Modulo. Journal of Automated Reasoning (JAR) **31**(1), 33–72 (2003)
10. Fisher, M.: An Open Approach to Concurrent Theorem Proving. Parallel Processing for Artificial Intelligence **3**, 80011–0 (1997)
11. Fitting, M.: First-Order Logic and Automated Theorem Proving. Springer (1990)
12. Fuchs, M., Wolf, A.: System Description: Cooperation in Model Elimination: CPTHEO. In: Conference on Automated Deduction (CADE). Lecture Notes in Computer Science (LNCS), vol. 1421, pp. 42–46. Springer (1998)
13. Hintikka, J.: Two Papers on Symbolic Logic: Form and Content in Quantification Theory and Reductions in the Theory of Types. Societas Philosophica, Acta philosophica Fennica **8**, 7–55 (1955)
14. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM **21**(8), 666–677 (1978)
15. Konrad, K.: HOT: A Concurrent Automated Theorem Prover Based on Higher-Order Tableaux. In: Theorem Proving in Higher Order Logics (TPHOLs). Lecture Notes in Computer Science (LNCS), vol. 1479, pp. 245–261. Springer (1998)
16. Korf, R.E.: Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. Artificial Intelligence **27**(1), 97–109 (1985)
17. Letz, R.: First-Order Tableau Methods. In: D'Agostino, M., Gabbay, D.M., Hähnle, R., Posegga, J. (eds.) Handbook of Tableau Methods, pp. 125–196. Springer (1999), iSBN 978-94-017-1754-0
18. Schumann, J., Letz, R.: PARTHEO: A High-Performance Parallel Theorem Prover. In: Conference on Automated Deduction (CADE). Lecture Notes in Computer Science (LNCS), vol. 449, pp. 40–56. Springer (1990)
19. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning (JAR) **59**(4), 483–502 (2017)
20. Tsoukalos, M.: Mastering Go: Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures, pp. 439–463. Packt Publishing Ltd. (2019)

21. Wu, C.H.: A Multi-Agent Framework for Distributed Theorem Proving. Expert Systems with Applications **29**(3), 554–565 (2005)