Is it Reasonable to Employ Agents in Automated Theorem Proving?*

Max Wisniewski[1] and Christoph Benzmüller[1]
[1]*Dept. of Mathematics and Computer Science, Freie Universität Berlin, Germany*
{*m.wisniewski, c.benzmüller*}*@fu-berlin.de*

Keywords: ATP, Multi-Agent Systems, Blackboard Architecture, Concurrent Reasoning

Abstract: Agent architectures and parallelization are, with a few exceptions, rarely to encounter in traditional automated theorem proving systems. This situation is motivating our ongoing work in the higher-order theorem prover Leo-III . In contrast to its predecessor – the well established prover LEO-II – and most other modern provers, Leo-III is designed from the very beginning for concurrent proof search. The prover features a multiagent blackboard architecture for reasoning agents to cooperate and to parallelize proof construction on the term, clause and search level.

1 INTRODUCTION

Leo-III (Wisniewski et al., 2014) – the successor of LEO-I (Benzmüller and Kohlhase, 1998) and LEO-II (Benzmüller et al., 2008) Leo-III – is a higher-order automated theorem prover currently under development at FU Berlin. It is supporting a polymorphically typed λ-calculus with nameless spine notation, explicit substitutions and perfect term sharing. On top of that it uses a multiagent blackboard architecture to parallelize the proof search.

The development of an automated theorem prover (ATP) often follows a common pattern. Significant time is spend on developing a new and more sophisticated calculus. The automated theorem prover is then designed as a sequential loop in which clause creating inference steps of the calculus, and normalization steps are performed alternately, just as in the famous Otter loop (McCune, 1990). The theorem prover may be augmented by means to collect and employ additional information. For example, in order to support its proof splitting technique, the AVATAR (Voronkov, 2014) prover maintains and uses information on models constructed by a SAT solver. Similarly, LEO-II incorporates a first-order logic automated theorem prover.

Although parallelization is heavily used and researched in computer science, there has not been much impact on the automated theorem proving community. A possible explanation is that parallelization introduces a range of additional challenges, whereas the implementation of a state of the art sequential prover is already challenging enough (and pays off easier). Even though various architectures have been proposed and studied (Bonacina, 2000), there are not many provers experimenting with parallelization (e.g. SETHEO derivatives SPTHEO (Suttner, 1997), PARTHEO (Schumann and Letz, 1990), or ROO (Lusk et al., 1992)). And, most of the existing implementations are only subsequent parallelizations of previously existing sequential theorem provers.

In Leo-III we are from the beginning developing the prover for parallel execution. Just as its predecessors, Leo-III aims at automating classical higher-order logic (Benzmüller and Miller, 2014). In comparison to the state of the art in first-order automated theorem proving, higher-order automated theorem provers are still much less sophisticated. In particular, there exists no single calculus yet which is empirically superior to all other calculi and the search space can be much bushier e.g. due to instantiations of propositional quantified variables.

Leo-III 's design is influenced by the Ω-ANTS system (Benzmüller et al., 2008), an agent-based command suggestion mechanism supporting the user in an interactive theorem proving environment. Each agent Ω-ANTS encompasses one calculus rule. Leo-III exploits this idea and eliminates the user from the system by letting the agents compete for the execution of their solutions.

The underlying architecture used in Leo-III has been made available as LEOPARD[2] (Wisniewski et al., 2015). This reusable system platform serves

*This work has been supported by the DFG under grant BE 2501/11-1 (Leo-III).

[2]https://github.com/cbenzmueller/LeoPARD

as a starting point to implement parallel, higher-order theorem provers. In particular, it aims at lowering the entry level for new developers in the area.

In the remainder of this position paper we address the following question: How reasonable is it to employ agents in automated theorem proving? Our focus is on higher-order logic and we outline a first implementation of a respective reasoner in LEOPARD.

2 HIGHER-ORDER LOGIC

Classical Higher-Order Logic is based on the simply typed λ-calculus[3] by Alonzo Church.

The syntax of the simply typed λ-calculus is defined in two layers – *types* τ,ν and *terms* s,t– over a set of constants Σ, variables $\mathcal{V}$ and base types T.

$$\begin{array}{rcll} \tau,\nu & ::= & t \in T & \text{(Base type)} \\ & | & \tau \to \nu & \text{(Abstraction type)} \end{array}$$

$$\begin{array}{rcll} s,t & ::= & X_\tau \in \mathcal{V} \mid c_\tau \in \Sigma & \text{(Variable / Constant)} \\ & | & (\lambda X_\tau\, s_\nu)_{\tau\to\nu} \mid (s_{\tau\to\nu}\, t_\tau)_\nu & \text{(Term abstr. / appl.)} \end{array}$$

Defining a logic in this syntax requires Σ to contain a complete logical signature. In this case we choose the signature $\{\vee_{o\to o\to o}, \neg_{o\to o}, \forall_{(\alpha\to o)\to o}\} \subseteq \Sigma$ for all $\alpha \in T$ and $\{o,i\} \subseteq T$ to be the type of booleans and individuals respectively. The remaining symbols can be defined as usual. A term s_o is called *formula*.

An example formula of this language is

$$\forall(\lambda A_{i\to o}.\,\exists(\lambda B_{i\to o}.\,\forall(\lambda X_i.\,\neg(AX)\vee(BX)))) \quad (1)$$

Here, $\exists(\lambda X.\,\varphi)$ stands for $\neg\forall(\lambda X.\,\neg\varphi)$. Interpreting A,B as characteristic predicates for sets this formula postulates, that each set has a superset. Note that by exploiting λ abstraction, there is no need to introduce an additional binding mechanism for quantifiers, provided the constants $\forall$ and $\exists$ are appropriately interpreted (see below).

To evaluate the truth value of a formula a model $\mathcal{M} = (\{D_\alpha\}_{\alpha\in T}, I)$ is introduced, where $D_\alpha \neq \emptyset$ is the *domain* of objects for type α and I is an interpretation assigning each symbol $c_\alpha \in \Sigma$ an object in D_α.

Given a substitution σ for variables we can define the valuation $[.]^\sigma_{\mathcal{M}}$:

$$\begin{array}{rcl} [X_\alpha]^\sigma_{\mathcal{M}} & = & \sigma(X_\alpha) \\ [c_\alpha]^\sigma_{\mathcal{M}} & = & I(c_\alpha) \\ [s_{\alpha\to\beta}t_\alpha]^\sigma_{\mathcal{M}} & = & [s_{\alpha\to\beta}]^\sigma_{\mathcal{M}}\,[t_\alpha]^\sigma_{\mathcal{M}} \\ [\lambda X_\alpha.\,s_\beta]^\sigma_{\mathcal{M}} & = & f_{\alpha\to\beta} \in D_{\alpha\to\beta} \\ & & \text{s.t. for each } z \in D_\alpha \text{ holds } fz = [s]^{\sigma\circ[z/X_\alpha]}_{\mathcal{M}} \end{array}$$

[3]Although Leo-III actually supports a polymorphic typed λ-calculus we will stick here to the simply typed λ-calculus for a better understanding.

$$\frac{[A[T_\gamma]]^\alpha \vee C \qquad [L_\gamma = R_\gamma]^T \vee D}{[A[R]]^\alpha \vee C \vee D \vee [T = L]^F}\ \text{para}$$

$$\frac{C \vee [A_{\gamma\to\beta} C_\gamma = B_{\gamma\to\beta} D_\gamma]^F}{C \vee [A = B]^F \vee [C = D]^F}\ \text{dec}$$

$$\vee^T\ \frac{C \vee [A \vee B]^T}{C \vee [A]^T \vee [B]^T} \qquad \frac{C \vee [A \vee B]^F}{C \vee [A]^F\ C \vee [B]^F}\ \vee^F$$

Figure 1: Selection of the rules of $\mathcal{EP}$

A formula s_o is called valid, iff $[s_o]^\sigma_{\mathcal{M}}$ evaluates to *true* under every substitution σ and every considered model $\mathcal{M}$. We will only consider Henkin models (where the function domains do not necessarily have to be full); for Henkin semantics sound and complete calculi exist (Benzmüller and Miller, 2014). For all the fixed signature $\{\vee_{o\to o\to o}, \neg_{o\to o}, \forall_{(\alpha\to o)\to o}\}$ we assume that I is fixed to the intended semantics and D_o only contains *true* and *false*. The choice of $I(\vee_{o\to o\to o})$ and $I(\neg_{o\to o})$ is obvious, and $I(\forall_{(\alpha\to o)\to o})$ is chosen to be a function, that applies the first argument to $\forall$ to every element of the domain D_α; *true* is returned if the result of this application leads to *true* in every case. With this interpretation of $\forall_{(\alpha\to o)\to o}$, it is easy to see that proposition is *valid*.

3 ATP

First- and higher-order ATPs cannot simply "compute" the truth value of a formula. Instead, ATPs "search" for proofs by employing sound and (ideally) complete proof calculi. One of the earliest proof calculi considered for automation is the *resolution calculus* (Robinson, 1965). As most other proof calculi for ATPs, the *resolution calculus* proves a formula by contradiction. Resolution based provers first transform the negated input problem into *conjunctive normal form (cnf)* and then they search for a proof using *resolution* and *factorization* rules. For first-order ATPs the resolution approach, explicitly one of its enhanced versions the *superposition calculus* (Bachmair and Ganzinger, 1994), seems to outperforms all other calculi. In higher-order ATPs the race for a "best" approach is much less settled.

We here take a closer look at a higher-order calculus – the paramodulation calculus $\mathcal{EP}$ (Benzmüller, 1999), which is a derivative of the resolution approach in the higher-order setting.

In Figure 1 a selection of rules of the calculus is displayed. The last two rules $\vee^T$ and $\vee^F$ are part of the clausification of the input problem. In an ATP

they can be exhaustively applied before all other rules or be interleaved with the rest. The *paramodulation rule* combines two clauses. It operates by replacing a subterm T in a literal of the first clause with the right-hand side of a positive equality literal $[L = R]^T$ in the second clause. This operation can of course only be performed, if L and T can be made equal, which is encoded in the literal $[T = L]^F$ and which can be understood as a unification constraint for the remaining clause.

Solving this unification constraint can be attempted eagerly. Generally, however, we allow to postpone its solution, since higher-order unification is not decidable. Instead unification constraints are handled in our approach by (unification) rules such as *dec*, which now become explicit calculus rules. *dec* decomposes an application and asserts equality of function and argument individually.

A sequential theorem prover based on such a calculus can be roughly implemented as follows:

```
while([] ∉ active_clauses)
  c <- active_clauses
  new = cnf(c)
  foreach(c1 in new)
    unify(c1)
  new += parmod(c,active_clauses)
  passiv_clauses += c
  active_clauses += new
```

We focus here on $\mathcal{EP}$ for two reasons: First the calculus of Leo-III , which is currently in development, will be an adaptation of the *superposition calculus* of first-order to higher-order. This calculus is closely related to the *paramodulation calculus* by restricting the newly created clauses by using ordering constraints.

The second reason is, that in this special calculus most of the steps are local or have at least very few dependencies outside of the inference. Hence, there exist many points to parallelize this loop, without introducing much overhead. Unifying the new clauses is a task completely mutually independent and hence, it can easily be parallelized. Same holds for the paramodulation step with all *active clauses*. With a bit more effort even the whole loop body can be parallelized, with a trick developed in ROO. This allows to select multiple clauses at once and to perform the loop body in parallel to itself.

Employing agents in other calculi is nonetheless possible, but there exist more constraints on the interactions. In a *tableaux calculus*, closing of a branch (i.e. unification) is a highly dependent operation since the implicit quantified variables are globally bound. On the other hand, expanding and exploring the proof tree is still possible to be performed in parallel by agents.

4 MULTIAGENT BLACKBOARD ARCHITECTURE

One important property of a multiagent system is, that no single agent can solve a given problem by itself (Weiss, 2013). This can be either achieved by a partial view on the available information for each agent or by limited capabilities of each agent. The idea of the architecture of Leo-III is to maintain all information on the current proof state globally available in a *blackboard*. The general principle of a *blackboard architecture* (Weiss, 2013) is that no expert (i.e. agent) hides information.

Just as in Ω-ANTS, the agents in Leo-III are autonomous implementations of inference or compound rules. No single agent in Leo-III will (usually) be able to solve an arbitrary given proof problem on its own.

Even though these agents are in the main focus implementing a multi-agent theorem prover, there are additional agents one could add to the system. First, we can incorporate external provers, just as done in LEO-II with first-order provers. Leo-III can even let higher-order provers compete and return the solution of the first prover to finish in order to find a result. Second, we can gather extra information from SAT solvers or model finders as AVATAR does or simply run different calculi in the same proof context. Lastly, the system allows to analyze itself during the run and adapt, for example the selection algorithm, to the current demand. Most importantly, all these task are inherently executed in parallel.

The main problem in designing a blackboard architecture is the access coordination of the agents to the blackboard. Two agents may read the same data. But the resulting state would be inconsistent, if two agents write to the same position at the same time. To solve this problem the action of an agent in Leo-III is designed as a *transaction*. As known from database systems, which suffer from the same problem, transactions are a common and elegant way to deal with updating inconsistencies. In the context of Leo-III these *transactions* are called *tasks*.

Leo-III operates in the following scheme (cf. Figure 2): Each agent perceives the proof state of the blackboard. Upon change of the proof state the agent decides on all actions he wants to execute and commits them to a scheduler as a *task*. The scheduler selects a non-conflicting set of *tasks* from all committed *tasks* and distributes them between the available processes. Finally the result of the transaction are inserted into the blackboard and a new iteration of this scheme is triggered.

Information distribution is performed through

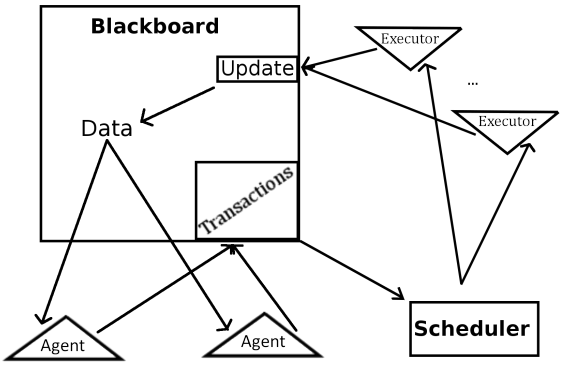

Figure 2: Parallelized Operation Loop of Leo-III

messages to the agents. Although inspired by LINDA (Gelernter et al., 1982) and JADE (Bellifemine et al., 2001), the messaging will not actively interact with the current execution of an agent. Agents handle a message by creating a new task, based on the content of the message. Just as in JADE the messages have a structure, to define a message type and its contents. Besides this there is no bigger protocol defined, since most messages are information messages of updated data in the blackboard.

The selection of non-conflicting *tasks* is a crucial part in the development of the prover. A wrong or one-sided selection of tasks could yield a non-complete implementation of the calculus, which might not only be unable to proof all valid formulas, but find no proofs at all.

An interesting side-effect of implementing agents through *tasks* is the inherent self-parallelization. Since each agent can commit several *tasks* at once, it is able to execute all of these *tasks* concurrently. Another feature of this agent implementation of a prover is we can play more loosely with the completeness of the system. As long as we can guarantee one complete calculus can execute eventually, we can adorn the system with incomplete, special purpose rules to enhance the proof search.

5 TASK SELECTION

The idea of the selection is to let the agents compete with each other which task should be executed. Each agent supports for its own *tasks* a ranking in which order it wants to execute the *tasks*. The ranking should reflect the benefit of a task to the proof search. The scheduler has then to select a set of non conflicting tasks to maximize the benefit to the proof search. This approach is similar to prominent heuristics as Shortest-Job-Next or Highest-Response-Ratio-Next for process schedulers with the addition, that selecting one task can kill others.

This problem is a generalization of the *combinatorical auction* (Nisan et al., 2007). In a *combinatorical auction* a bidder tries to achieve a set of items. He wants either all of the items in the set or non of them. The auctioneer has to assign the items to the bidders, such that no item is shared by two bidders and the revenue is maximal. The *task auction* we play in our scheduler allows two kinds of items in the bidders set: *sharable items* that are only read, and *private items* that are also written. It is easy to see, that *combinatorical auctions* can be reduced to *task auctions*.

As in many scheduling problems the *combinatorical auction* is *NP*-complete. Luckily there exists a fast approximation algorithm for *combinatorical auctions*, that can be easily adapted for the *task auction*.

```
auction(t : 2^Task):
  take = ∅
  while (t ≠ ∅):
    i = argmax_{x∈t} |x.writeSet ∪ x.readSet|
    take = i :: take
    t = {x ∈ t | ¬conflict(x,i)}
    t = {y | x ∈ t, y.r = x.r \ i.r}
return take
```

For n bidders and k distinct items in t the algorithm `auction` is a $\sqrt{k}$-approximation of the *task auction* and runs in $O(k^2n^2)$ (Wisniewski, 2014).

The system runs by iterating this scheduling algorithm. The runtime suggest to restrain the amount of committed tasks per agent. In Leo-III each agent can make its own preselection of tasks to keep the burden on the scheduler to a minimum. To ensure that each agent will eventually be able to execute, a natural aging is implemented: Each round an agent is granted an income, which he can spend on its *tasks*. Hence an agent that has been idle for a long time will have a better chance to acquire one of its *tasks*.

One open problem for this approach is *priority inversion*. Since we create a *task* only if it is possible to execute, we are not able to track any dependencies that might block a task. This problem can be solved as soon as the ranking of the *tasks* is supervised by the blackboard. As mentioned earlier we want to build agents controlling the selection mechanism and adapting it to the current situation. This mechanism can be exploited by the agents to boost the importance of their counterparts to push the importance of their tasks. One could even think about a donation system between the agents to model this behavior in the auction game.

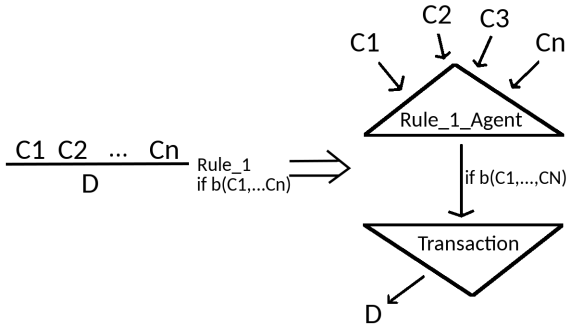

Figure 3: Transforming a rule into an agent.

6 RULE-BASED AGENTS

For a first test of the agent approach for theorem proving we implemented the *paramodulation* calculus from Section 3 with the described agent architecture. As described in Figure 3 for each agent we created an agent. This agents look at the blackboard and upon insertion of a new clause C_1 search for compatible clauses. In this first, simple test the blackboard only offered a set of clauses. Hence the agent has to search all existing clauses for a partners $C_2, \ldots, C_n$. With these partners the inference constraints $b(C_1, \ldots, C_n)$ is tested. The calculation of the result D is delayed as a transaction and send to the scheduler.

Applying this structure to the inference rules of $\mathcal{EP}$ we obtain agents of the following form:

```
OrF_Agent {
 OnInsert(C) if(containsOrF(C)) ->
  C', A, B <- decomposeOrF(C)
  OrFTask(C', A ,B)
}

OrFTask(C', A, B) {
 execute ->
  insert( [A]^F :: C' )
  insert( [B]^F :: C' )
}
```

The agents can react on insertion triggers to look at every inserted data, in this case a clause. In simple rules as this clausification no further cooperation with the blackboard is needed.

As already mentioned in this first implementation this happened by iterating over all existing, active clauses. But the blackboard allows it easily to add specialized search data structures for each agent to improve on the time for searching partners. To still model part of the sequential loop behavior we introduced the passive/active status by adding a selection timer to all clauses considered for paramodulation.

The bid assigned to each *task* favored a small amount of literals. The bid for a *task* is the wealth of the agent divided by the amount of literals in the task. Although there is not yet a completeness result for this implementation, the idea to increasingly augment the proof state beginning with the smallest clauses is one of the promising options. Since this process enumerates all valid clauses implied by the initial clauses, the proof will eventually be found.

Astonishingly, this implementation yields an already powerful prover. For example, we were able to proof some of the Boolean properties of sets from the TPTP (`SET014^5`, `SET027^5`, `SET067^1`, etc.) including the distributivity law of union and intersection, and even the *surjective cantor theorem*. The later one states that there cannot exist a surjective function from a set to its superset or, in other words, a superset is strictly greater then the underlying set. Interestingly, most of these theorems are very difficult for first-order provers to solve. Anyhow, our initial experiments indicate that a multiagent based proof search approach is pragmatically feasible. More experiments are clearly needed to assess the benefits and limits of the approach.

7 FURTHER WORK

As soon as the general calculus for Leo-III is fixed we will start experimenting on the granularity of the rules. For the test implementation of $\mathcal{EP}$ we choose to implement every inference rule as a single agent. Commonly in the implementation we can identify reasonable sequential workflow, e.g. a clause inferred in a unification should be immediately simplified. The first approach of agentifying the *inference rules* will hence be extended to agentifying a tactic level of compound rules.

Orthogonal to experimenting on the Leo-III calculus applying agents to other calculi is possible. As mentioned other calculi as *tableaux calculus* or *natural deduction* are more globally depended, but one could look into a parallel approach to these calculi. Even more interesting is the combination of different calculi. This scenario is interesting, if the calculi share the same search space, and hence can cooperate.

LEO-II uses first-order ATPs as subprovers. Periodically, after conducting some higher-order specific inferences, (part of) the proof context is translated into a first-order representation and send to an external prover. This is a direction, we again want to develop Leo-III into. The agent architecture allows us to run multiple instances of the subprovers in parallel. Either with different proof modi (flag settings) or varying proof obligations.

Concerning the auction scheduler we still have to

deal with priority inversion. As presented we can augment the auction scheduler with a donation system allowing high priority agents to donate to low priority, dependent agents. This augmentation should be tested for performance and practicality.

8 CONCLUSION

We initially posed the question whether employing agents in higher-order automated theorem proving is reasonable. We have then provided some first evidence that applying an agent architecture to higher-order automated theorem proving is not only possible, but that even straightforward implementations may yield promising systems in practice. First experiments were successful, and for the open problems multiple directions for further developments exist.

Besides the parallelization of a calculus, the agent-based approach benefits from a high flexibility. It is easily possible to add and remove agents from an inference, to a tactic up to a complete theorem prover layer. The high amount of parameters in ATPs and the vast amount of small single tactics in interactive provers indicate a great potential in the agent technology. In traditional systems different settings are tested sequentially and many shared tasks (e.g. normalization) are duplicated and executed numerous times. Running in a shared setting can reduce this number significantly.

ACKNOWLEDGMENTS We would like to thank all members of the Leo-III project.

REFERENCES

Bachmair, L. and Ganzinger, H. (1994). Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comp.*, 4(3):217–247.

Bellifemine, Fabio and Rimassa, Giovanni (2001), Developing Multi-agent Systems with a FIPA-compliant Agent Framework. *J. Softw. Pract. Exper.*, 31, pages 103–128, John Wiley & Sons.

Benzmüller, C. (1999). Extensional higher-order paramodulation and RUE-resolution. In *CADE*, number 1632 in LNCS, pages 399–413. Springer.

Benzmüller, C. and Kohlhase, M. (1998). LEO – a higher-order theorem prover. In *CADE*, number 1421 in LNCS, pages 139–143. Springer.

Benzmüller, C. and Miller, D. (2014). Automation of higher-order logic. In Siekmann, J., Gabbay, D., and Woods, J., eds., *Handbook of the History of Logic, Vol. 9 – Logic and Computation*. Elsevier.

Benzmüller, C., Theiss, F., Paulson, L., and Fietzke, A. (2008). LEO-II - a cooperative automatic theorem prover for higher-order logic. In *IJCAR*, volume 5195 of *LNCS*, pages 162–170. Springer.

Benzmüller, C., Sorge, V., Jamnik, M., and Kerber, M. (2008). Combined reasoning by automated cooperation. *J. Appl. Log.*, 6(3):318 – 342.

Bonacina, M. (2000). A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):223–257.

Gelernter, David and Bernstein, Arthur J./1982) Distributed Communication via Global Buffer. Proceedings of the First ACM SIGACT-SIGOPS, *PODC '82*, pages 10–18. ACM.

Lusk, E. L., McCune, W., and Slaney, J. K. (1992). Roo: A parallel theorem prover. In *CADE*, volume 607 of *LNCS*, pages 731–734. Springer.

McCune, W. (1990). OTTER 2.0. In *CADE*, pages 663–664.

Nisan, N., Roughgarden, T., Tardos, E., and Vazirani, V. V. (2007). *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41.

Schumann, J. and Letz, R. (1990). Partheo: A high-performance parallel theorem prover. In *CADE*, volume 449 of *LNCS*, pages 40–56. Springer.

Suttner, C. B. (1997). Sptheo - a parallel theorem prover. *J. Autom. Reason.*, 18(2):253–258.

Voronkov, A. (2014). AVATAR: the architecture for first-order theorem provers. In *CAV*, volume 8559 in LNCS, pages 696–710. Springer.

Weiss, G., editor (2013). *Multiagent Systems*. MIT Press.

Wisniewski, M. (2014). Agent-based blackboard architecture for a higher-order theorem prover. Master's thesis, Freie Universität Berlin.

Wisniewski, M., Steen, A., and Benzmüller, C. (2014). The Leo-III project. In Bolotov, A. and Kerber, M., editors, *Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38.

Wisniewski, M., Steen, A., and Benzmüller, C. (2015). Leopard - A generic platform for the implementation of higher-order reasoners. In Kerber, M., et al., editors, *CICM*, volume 9150 of *LNCS*, pages 325–330. Springer.