UNIVERSITÄT DES SAARLANDES
FACHBEREICH INFORMATIK
D-66041 SAARBRÜCKEN
GERMANY
WWW: http://www.ags.uni-sb.de/

SEKI Report

Resource Adaptive Agents in Interactive Theorem Proving

Christoph Benzmüller and Volker Sorge
{chris|sorge}@ags.uni-sb.de
http://www.ags.uni-sb.de/~{chris|sorge}

SEKI Report SR-99-02

Abstract

We introduce a resource adaptive agent mechanism which supports the user in interactive theorem proving. The mechanism, an extension of [4], uses a two layered architecture of agent societies to suggest appropriate commands together with possible command argument instantiations. Experiments with this approach show that its effectiveness can be further improved by introducing a resource concept. In this paper we provide an abstract view on the overall mechanism, motivate the necessity of an appropriate resource concept and discuss its realization within the agent architecture.

1 Introduction

Interactive theorem provers have been developed to overcome the shortcomings of purely automatic systems by enabling the user to guide the proof search and by directly importing expert knowledge into the system. For large proofs, however, this task becomes difficult if the system does not provide a sophisticated mechanism to suggest possible next steps in order to minimize the necessary interactions.

Suggestion mechanisms in interactive theorem proving systems such as HOL [7], TPS [2] or ΩMEGA [3] are rather limited in their functionality as they usually

(i) use inflexible sequential computation strategies,
(ii) do not have anytime character,
(iii) do not work steadily and autonomously in the background of a system, and
(iv) do not exhaustively use available computation resources

In order to overcome these limitations, [4] proposed — within the context[1] of tactical theorem proving based on ND-calculus [5] — a new, flexible support mechanism with anytime character. It suggests commands, applicable in the current proof state — more precisely commands that invoke some ND-rule or tactic — together with a suitable argument instantiations. It is based on two layers of societies of autonomous, concurrent agents which steadily work in the background of a system and dynamically update their computational behavior to the state of the proof and/or specific user queries to the suggestion mechanism. By exchanging relevant results via blackboards the agents cooperatively accumulate useful command suggestions which can then be heuristically sorted and presented to the user.

[1] We want to point out that this mechanism is in no way restricted to a specific logic or calculus and can easily be adapted to other interactive theorem contexts as well.

A first Lisp-based implementation of the support mechanism in the ΩMEGA-system yielded promising results. However as we could not exploit concurrency in the Lisp implementation experience showed that we had to develop a resource adapted concept for the agents in order to allow for efficient suggestions even in large examples. In the current reimplementation of major parts of ΩMEGA and the command suggestion mechanism in Oz, a concurrent constraint logic programming language [12], the first impression is that the use of concurrency provides a good opportunity to switch from a static to a dynamic, resource adaptive control of the mechanism's computational behavior. Thereby, we can exploit both knowledge on the prior performance of the mechanism as well as knowledge on classifying the current proof state and single agents in order to distribute resources.

In this paper we first sketch our two layered agent mechanism and introduce the static approach of handling resources. We then extend this approach into a resource adaptive[2] one where the agents monitor their own contributions and performance in the past in order to estimate their potential contribution and performance within the next step and where the adequacy of all agents computations with respect to the complexity of the current proof goal is analyzed by a special *classification agent*. Thus, the agents have a means to decide whether or not they should pursue their own intentions in a given state. Moreover, the overall mechanism can dynamically change resource allocations thereby enabling or disabling computations of single agents or complete societies of agents. And finally, the classification agent sends deactivation signals to single agents or agent societies as soon as evidence or definite knowledge is available that these agents can currently not compute any appropriate contributions (e.g. when an agent belongs to a first-order command whereas the current proof goal can be classified as propositional).

2 Reference Example

The following example will be used throughout this paper: $(p_{o\to o}\ (a_o \wedge b_o)) \Rightarrow (p\ (b \wedge a))$. Whereas it looks quite simple at a first glance this little higher-order (HO) problem cannot be solved by most automatic HO theorem provers known to the authors, since it requires the application of

[2]In this paper we adopt the notions of resource adapted and resource adaptive as defined in [13], where the former means that agents behave with respect to some initially set resource distribution. According to the latter concept agents have an explicit notion of resources themselves, enabling them to actively participate in the dynamic allocation of resources.

the extensionality principles which are generally not built-in in HO theorem proving systems. Informally this example states: If the truth value of $a \wedge b$ is element of the set p of truth values, then $b \wedge a$ is also in p. In the mathematical assistant ΩMEGA [3], which employs a variant of Gentzen's natural deduction calculus (ND) [5] enriched by more powerful proof tactics, the following proof can easily be constructed interactively[3]:

L_1	$(L_1) \vdash$	$(p\ (a \wedge b))$	Hyp
L_4	$(L_1) \vdash$	$(b \wedge a) \Leftrightarrow (a \wedge b)$	OTTER
L_3	$(L_1) \vdash$	$(b \wedge a) = (a \wedge b)$	$\Leftrightarrow 2= : (L_4)$
L_2	$(L_1) \vdash$	$(p\ (b \wedge a))$	$=_{\text{subst}}: (\langle 1 \rangle)(L_1 L_3)$
C	$() \vdash$	$(p\ (a \wedge b)) \Rightarrow (p\ (b \wedge a))$	$\Rightarrow_I : (L_2)$

The idea of the proof is to show that the truth value of $a \wedge b$ equals that of $b \wedge a$ (lines L_3 and L_4) and then to employ equality substitution (line L_2). The equation $(b \wedge a) = (a \wedge b)$, i.e. by boolean extensionality the equivalence $(b \wedge a) \Leftrightarrow (a \wedge b)$ is proven here with the first-order prover OTTER [9] and the detailed subproof is hidden at the chosen presentation level. Our agent mechanism is able to suggest all the single proof steps together with the particular parameter instantiations to the user. In this paper we use parts of this example to demonstrate the working scheme of the mechanism and to motivate the use of resources.

3 Suggesting Commands

The general suggestion mechanism is based on a two layered agent architecture displayed in Fig. 1 which shows the actual situation after the first proof step in the example, where the backward application of $\Rightarrow_I$ introduces the hypothesis line L_1 and L_2 as the new open goal. The task of the bottom layer of agents (cf. the lower part of Fig. 1) is to compute possible argument instantiations for the provers commands in dependence of the dynamically changing partial proof tree. The task of the top layer (cf. the upper part of Fig. 1) is to collect the most appropriate suggestions from the bottom layer, to heuristically sort them and to present them to the user.

The bottom layer consists of societies of argument agents where each society belongs to exactly one command associated with a proof tactic[4] (cf. below). On the one hand each argument agent has its own intention, namely

[3] Linearized ND proofs are presented as described in [1]. Each proof line consists of a label, a set of hypotheses, the formula and a justification.

[4] For our approach it is not necessary to distinguish between proof rules, proof tactics and proof methods and therefore we just use the phrase "tactic" within this article.

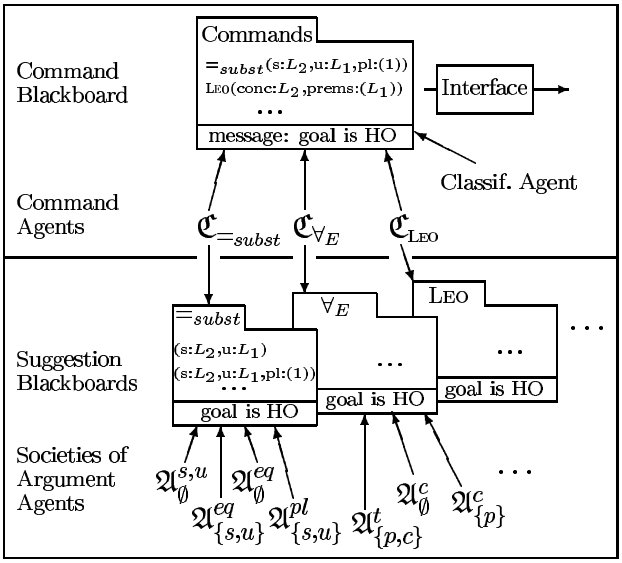

Figure 1: The two layered suggestion mechanism.

to search in the partial proof for a proof line that suits a certain specification. On the other hand argument agents belonging to the same society also pursue a common goal, e.g. to cooperatively compute most complete argument suggestions (cf. concept of PAI's below) for their associated command. Therefore the single agents of a society exchange their particular results via a suggestion blackboard and try to complete each others suggestions.

The top layer consists of a single society of command agents which steadily monitor the particular suggestion blackboards on the bottom layer. For each suggestion blackboard there exists one command agent whose intention is to determine the most complete suggestions and to put them on the command blackboard.

The whole distributed agent mechanism runs always in the background of the interactive theorem proving environment thereby constantly producing command suggestions that are dynamically adjusted to the current proof state. At any time the suggestions on the command blackboard are monitored by an interface component which presents them heuristically sorted to the user via a graphical user interface. As soon as the user executes a command the partial proof is updated and simultaneously the suggestion and command blackboards are reinitialized.

3.1 Partial Argument Instantiations (PAI)

The data that is exchanged within the blackboard architecture heavily depends on a concept called a *partial argument instantiation (PAI)* of a command. In order to clarify our mechanism we need to introduce this concept in detail.

In an interactive theorem prover such as ΩMEGA or HOL one has generally one command associated with each proof tactic that invokes the application of this tactic to a set of proof lines. In ΩMEGA these tactics have a fixed *outline*, i.e. a set of premise lines, conclusion lines and additional parameters, such as terms or term-positions. Thus the general instance of a tactic $\mathcal{T}$ can be formalized in the following way:

$$\frac{P_1 \cdots P_l}{C_1 \cdots C_m}\ \mathcal{T}(Q_1 \cdots Q_n),$$

where we call the P_i, C_j, Q_k the *formal arguments* of the tactic $\mathcal{T}$ (we give an example below).

We can now denote the command t invoking tactic $\mathcal{T}$ formally in a similar fashion as

$$\frac{p_{i_1} \cdots p_{i_{l'}}}{c_{j_1} \cdots c_{j_{m'}}}\ t(q_{k_1} \cdots q_{k_{n'}}),$$

where the formal arguments p_i, c_j, q_k of t correspond to a subset of the formal arguments of the tactic. To successfully execute the command some, not necessarily all, formal arguments have to be instantiated with *actual arguments*, e.g. proof lines. A set of pairs relating each formal argument of the command to an (possibly empty) actual argument is called a *partial argument instantiation (PAI)*.

We illustrate the idea of a PAI using the tactic for *equality substitution* $=_{Subst}$ and its corresponding command $=\!Subst$ as an example.

$$\frac{\Phi[x] \quad x = y}{\Phi'[y]} =_{Subst}(P^*) \quad \rightarrow \quad \frac{u \quad eq}{s} =\!Subst(pl)$$

Here $\Phi[s]$ is an arbitrary higher order formula with at least one occurrence of the term s, P^* is a list of term-positions representing one or several occurrences of s in Φ, and $\Phi'[t]$ represents the term resulting from replacing s by t at all positions P^* in Φ. *u, eq, s* and *pl* are the corresponding formal arguments of the command associated with the respective formal arguments of the tactic. We observe the application of this tactic to line L_2 of our example:

$$\begin{array}{llll} L_1 & (L_1) & \vdash & (p\ (a \wedge b)) \quad \text{Hyp} \\ & & \cdots & \\ L_2 & (L_1) & \vdash & (p\ (b \wedge a)) \quad \text{Open} \end{array}$$

As one possible PAI for *=Subst* we get the set of pairs $(u{:}L_1, eq{:}\epsilon, s{:}L_2, pl{:}\epsilon)$, where ϵ denotes the empty or unspecified actual argument. We omit writing pairs containing ϵ and, for instance, write the second possible PAI of the above example as $(u{:}L_1, s{:}L_2, pl{:}(\langle 1 \rangle))$. To execute *=Subst* with the former PAI the user would have to at least provide the position list, whereas using the latter PAI results in the line L_3 of the example containing the equality.

3.2 Argument Agents

The idea underlying our mechanism to suggest commands is to compute PAIs as complete as possible for each command, thereby gaining knowledge on which tactics can be applied combined with which argument instantiations in a given proof state.

The main work is done by the societies of cooperating *Argument Agents* at the bottom layer (cf. Fig. 1). Their job is to retrieve information from the current proof state either by searching for proof lines appropriate to the agents specification or by computing some additional parameter (e.g. a list of sub-term positions) with already given information. Sticking to our example we can informally specify the agents $\mathfrak{A}^{u,s}_{\emptyset}$, $\mathfrak{A}^{eq}_{\emptyset}$, $\mathfrak{A}^{eq}_{\{u,s\}}$, and $\mathfrak{A}^{pl}_{\{u,s\}}$ for the *=Subst* command (cf. [4] for a formal specification):

$$\begin{aligned} \mathfrak{A}^{s,u}_{\emptyset} &= \left\{ \begin{array}{l} \text{find open line and a support line that dif-} \\ \text{fer only wrt. occurences of a single proper} \\ \text{subterm} \end{array} \right\} \\ \mathfrak{A}^{eq}_{\emptyset} &= \{\text{find line with an equality}\} \\ \mathfrak{A}^{eq}_{\{u,s\}} &= \{\text{find equality line suitable for } s \text{ and } u\} \\ \mathfrak{A}^{pl}_{\{u,s\}} &= \{\text{compute positions where } s \text{ and } u \text{ differ}\} \end{aligned}$$

The attached superscripts specify the formal arguments of the command for which actual arguments are computed, whereas the indices denote sets of formal arguments that necessarily have to be instantiated in some PAI, so that the agent can carry out its own computations. For example agent $\mathfrak{A}^{eq}_{\{u,s\}}$ only starts working when it detects a PAI on the blackboard where actual arguments for u and s have been instantiated. On the contrary $\mathfrak{A}^{eq}_{\emptyset}$ does

not need any additional knowledge in order to pursue its task to retrieve an open line containing an equality as formula.

The agents themselves are realized as autonomous processes that concurrently compute their suggestions whereas they are triggered by the PAIs on the blackboard, i.e. the results of other agents of their society. For instance both agents $\mathfrak{A}^{eq}_{\{u,s\}}$ and $\mathfrak{A}^{pl}_{\{u,s\}}$ would simultaneously start their search as soon as $\mathfrak{A}^{s,u}_{\emptyset}$ has returned a result. The agents of one society cooperate in the sense that they activate each other (by writing new PAIs to the blackboard) and furthermore complete each others suggestions.

Conflicts between agents do not arise, as agents that add actual parameters to some PAI always write a new copy of the particular PAI on the blackboard, thereby keeping the original less complete PAI intact. The agents themselves watch their suggestion blackboard (both PAI entries and additional messages, cf. 4) and running agents terminate as soon as the associated suggestion blackboard is reinitialized, e.g. when a command has been executed by the user.

The left hand side of Fig. 1 illustrates our above example: The topmost suggestion blackboard contains the two PAIs: $(u{:}L_1, s{:}L_2)$ computed by agent $\mathfrak{A}^{s,u}_{\emptyset}$ and $(u{:}L_1, s{:}L_2, pl{:}(\langle 1\rangle))$ completed by agent $\mathfrak{A}^{eq}_{\{u,s\}}$.

3.3 Command Agents

In the society of *command agents* every agent is linked to a command and its task is to initialize and monitor the associated suggestion blackboards. Its intention is to select among the entries of the associated blackboard the most complete and appropriate one and to pass it, enriched by the corresponding command name to the command blackboard. That is, as soon as a PAI is written to the related blackboard that has at least one actual argument instantiated, the command agent suggests the command as applicable in the current proof state, providing also the PAI as possible argument instantiations. It then updates this suggestion, whenever a *better* PAI has been computed. In this context *better* generally means a PAI containing more actual arguments. In the case of our example the current PAI suggested by command agent $\mathfrak{C}_{=Subst}$ is $(u{:}L_1, s{:}L_2, pl{:}(\langle 1\rangle))$.

These suggestions are accumulated on a *command blackboard*, that simply stores all suggested commands together with the proposed PAI, continuously handles updates of the latter, sorts and resorts the single suggestions and provides a means to propose them to the user. In the case of the ΩMEGA-system this is achieved in a special command suggestion window within the graphical user interface $\mathcal{LOUI}$ [10]. The sorting of the suggestions is done

according to several heuristic criteria, one of which is that commands with fully instantiated PAIs are always preferred as their application may conclude the whole subproof.

3.4 Experiences

The mechanism sketched here has some obvious advantages. Firstly, it has anytime character and does not waste system resources, as it constantly works in a background process and steadily improves its suggestions wrt. its heuristic sorting criteria. Secondly, in contrast to conventional command and argument suggestion mechanisms, our approach provides more flexibility in the sense that agents can be defined independently from the actual command and the user can choose freely among several given suggestions. Thirdly, we can employ concurrent programming techniques to parallelize the process of computing suggestions.

Unfortunately, computations of single agents themselves can be very costly. Reconsider the agents of command *=Subst*: In ΩMEGA we have currently 25 different argument agents defined for *=Subst* where some are computationally highly expensive. For example, while the agent $\mathfrak{A}_{\emptyset}^{eq}$ only tests head symbols of formulas during its search for lines containing an equality and is therefore relatively inexpensive, the agent $\mathfrak{A}_{\emptyset}^{u,s}$ performs computationally expensive matching operations. In large proofs agents of the latter type might not only take a long time before returning any useful result, but also will absorb a fair amount of system resources, thereby slowing down the computations of other argument agents.

[4] already tackles this problem partially by introducing a *focusing technique* that explicitly partitions a partial proof into subproblems in order to guide the search of the agents. This focusing technique takes two important aspects into account: (i) A partial proof often contains several open subgoals and humans usually focus on one such subgoal before switching to the next. (ii) Hypotheses and derived lines belonging to an open subgoal are chronologically sorted where the interest focuses on the more recently introduced lines. Hence, the agents restrict their search to the actual subgoal (*actual focus*) and guide their search according to the chronological order of the proof lines.

4 Resource Adapted Approach

Apart from this we use a concept of *static complexity ratings* where a rating is attached to each argument and each command agent, that roughly

reflects the computational complexity involved for its suggestions. A *global complexity value* can then be adjusted by the user permitting to suppress computations of agents, whose ratings are larger than the specified value. Furthermore commands can be completely excluded from the suggestion process. For example, the agent $\mathfrak{A}_{\emptyset}^{u,s}$ has a higher complexity rating than $\mathfrak{A}_{\emptyset}^{eq}$ from the *=Subst* example, since recursively matching terms is generally a harder task than retrieving a line containing an equality. The overall rating of a command agent is set to the average rating of its single argument agents. This rating system increased the effectiveness of the command suggestions for the simulation of the architecture in LISP. (In the simulation a user was always forced to wait for all active agents to finish their computations.) However, the system is very inflexible as ratings are assigned by the programmer of a particular agent and cannot be adjusted by the user at runtime. Furthermore, the ratings do not necessarily reflect the actual relative complexity of the agents.

Shifting to the Oz implementation the latter problem no longer arises since a user can interrupt the suggestion process by choosing a command at any time without waiting for all possible suggestions to be made. In the Oz implementation every agent is an independent thread. It either quits its computations regularly or as soon as it detects that the blackboard it works for has been reinitialized, i.e. when the user has executed a command. It then performs all further computations with respect to the reinitialized blackboard.

However, with increasing size of proofs some agents never have the chance to write meaningful suggestions to a blackboard. Therefore, these agents should be excluded from the suggestion process altogether, especially if their computations are very costly which deprives other agents of resources. But while in the sequential simulation in LISP it was still possible for a user to monitor which agent is computationally very expensive and to subsequently adjust the global complexity value accordingly, this is no longer possible in the concurrent setting as the user in general is not aware of which agents produced useful results. Furthermore, the user should be as far as possible spared from fine-tuning a mechanism designed for his/her own support.

5 Resource Adaptive Approach

In this section we extend the resource adapted approach into a resource adaptive one. While we retain the principle of activation/deactivation by comparing the particular complexity ratings of the argument agents with

the overall deactivation threshold, we now allow the individual complexity ratings of argument agents to be dynamically adjusted by the system itself. Furthermore, we introduce a special classification agent which analyses and classifies the current proof goal in order to deactivate those agents which are not appropriate wrt. the current goal.

5.1 Dynamic Adjustment of Ratings

The dynamic adjustment takes place on both layers: On the bottom layer we allow the argument agents to adjust their own ratings by reflecting their performance and contributions in the past. On the other hand the command agents on the top layer adjust the ratings of their associated argument agents. This is motivated by the fact that on this layer it is possible to compare the performance and contribution of agent societies of the bottom layer.

Therefore, agents need an explicit concept of resources enabling them to communicate and reason about their performance. The communication is achieved by propagating resource informations from the bottom to the top layer and vice versa via the blackboards. The actual information is gathered by the agents on the bottom layer of the architecture. Currently the argument agents evaluate their effectiveness with respect to the following two measures:

1. the absolute cpu time the agents consume, and
2. 'the patience of the user', before executing the next command.

(1) is an objective measure that is computed by each agent at runtime. Agents then use these values to compute the average cpu time for the last n runs and convert the result into a corresponding complexity rating.

Measure (2) is rather subjective which expresses formally the ability of an agent to judge whether it ever makes contributions for the command suggesting process in the current proof state. Whenever an agent returns from a computation without any new contribution to the suggestion blackboard, or even worse, whenever an agent does not return before the user executes another command (which reinitializes the blackboards), the agent receives a penalty that increases its complexity rating. Consequently, when an agent fails to contribute several times in a row, its complexity rating quickly exceeds the deactivation threshold and the agent retires.

Whenever an argument agent updates its complexity rating this adjustment is reported to the corresponding command agent via a blackboard

entry. The command agent collects all these entries, computes the average complexity rating of his argument agents, and reports the complete resource information on his society of argument agents to the command blackboard. The command blackboard therefore steadily provides information on the effectiveness of all active argument agents, as well as information on the retired agents and an estimation of the overall effectiveness every argument agent society.

An additional *resource agent* uses this resource information in order to reason about a possibly optimal resource adjustment for the overall system, taking the following criteria into account:

- Assessment of absolute cpu times.
- A minimum number of argument agents should always be active. If the number of active agents drops below this value the global complexity value is readjusted in order to reactivate some of the retired agents.
- Agent societies with a very high average complexity rating and many retired argument agents should get a new chance to improve their effectiveness. Therefore the complexity ratings of the retired agents is lowered beneath the deactivation threshold.
- In special proof states some command agent (together with their argument agents) are excluded. For example, if a focused subproblem is a propositional logic problem, commands invoking tactics dealing with quantifiers are needless.

Results from the resource agent are propagated down in the agent society and gain precedence over the local resource adjustments of the single agents.

5.2 Informed Activation & Deactivation

Most tactics in an interactive theorem prover are implicitly associated with a specific logic (e.g. propositional, first-order, or higher -order logic) or even with a specific mathematical theory (e.g. natural numbers, set theory). This obviously also holds for the proof problems examined in a mathematical context. Some systems – for instance the ΩMEGA-System – do even explicitly maintain respective knowledge by administering all rules, tactics, etc. as well as all proof problems within a hierarchically structured theory database. This kind of classification knowledge can fruitfully be employed by our agent mechanism to activate appropriate agents and especially to deactivate non-appropriate ones. Even if a given proof problem can not be associated

with a very restrictive class (e.g. propositional logic) from the start, some of the subproblems subsequently generated during the proof probably can. This can be nicely illustrated with our example: The original proof problem belonging to higher-order logic gets transformed by the backward application of $\Rightarrow_I$, $=_{\text{Subst}}$ and $\equiv 2=$ into a very simple propositional logic problem (cf. line L_4). In this situation agents associated with a command from first- or higher-order logic (like $=Subst$, $\forall E$, or LEO[5]) should be disabled.

Therefore we add a classification agent to our suggestion mechanism whose only task is to investigate each new subgoal in order to classify it wrt. to the known theories or logics. As soon as this agent is able to associate the current goal with a known class or theory it places an appropriate entry on the command blackboard (cf. "HO" entry in Fig. 1). This entry is then broadcasted to the lower layer suggestion blackboards by the command agents where it becomes available to all argument agents. Each argument agent can now compare its own classification knowledge with the particular entry on the suggestion blackboard and decide whether it should perform further computations within the current system state or not.

The motivation for designing the subgoal classifying component as an agent itself is clear: It can be very costly to examine whether a given subgoal belongs to a specific theory or logic. Therefore this task should be performed concurrently by the suggestion mechanism and not within each initialization phase of the blackboard mechanism. Whereas our current architecture provides one single classification agent only, the single algorithms and tests employed by this component can generally be further distributed by using a whole society of classification agents.

6 Conclusion

In this paper we reported on the extension of the concurrent command suggestion mechanism [4] to a resource adaptive approach. The resources that influence the performance of our system are: (i) The available computation time and memory space. (ii) Classification knowledge on the single agents and the agent societies. (iii) Criteria and algorithms available to the classification agent.

Our approach can be considered as an instance of a boundedly rational system [13, 11]. The work is also related to [6] which presents an abstract resource concept for multi-layered agent architectures. [8] describes a successful application of this framework within the Robocup simulation.

[5] A higher-order theorem prover integrated to ΩMEGA.

Consequently some future work should include a closer comparison of our mechanism with this work.

The presented extensions are currently implemented and analyzed in ΩMEGA. This might yield further possible refinements of the resource concepts to improve the performance of the mechanism. Another question in this context is, whether learning techniques can support our resource adjustments on the top layer, as it seems to be reasonable that there even exist appropriate resource patterns for the argument agents in dependence of the focused subproblem.

Acknowledgements We thank Serge Autexier and Christoph Jung for stimulating discussions.

References

[1] P. B. Andrews. *An Introduction To Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, San Diego, CA, USA, 1986.

[2] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A Theorem Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

[3] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMega: Towards a Mathematical Assistant. In W. McCune, editor, *Proceedings of CADE–14*, volume 1249 of *LNAI*, pages 252–255. Springer Verlag, 1997.

[4] Christoph Benzmüller and Volker Sorge. A Blackboard Architecture for Guiding Interactive Proofs. In F. Giunchiglia, editor, *Proceedings of AIMSA'98*, number 1480 in LNAI, pages 102–114. Springer Verlag, 1998.

[5] G. Gentzen. Untersuchungen über das Logische Schließen I und II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[6] C. Gerber and C. G. Jung. Resource management for boundedly optimal agent societies. In *Proceedings of the ECAI98 Workshop on Monitoring and Control of Real-Time Intelligent Systems*, pages 23–28, 1998.

[7] M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[8] C. G. Jung. Experimenting with layered, resource-adapting agents in the robocup simulation. In *Proc. of the ROBOCUP'98 Workshop*, 1998.

[9] William McCune and Larry Wos. Otter CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE-13 Automated Theorem Proving System Competition.

[10] J. Siekmann, S. M. Hess, C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, M. Kohlhase, K. Konrad, E. Melis, A. Meier, and V. Sorge. LΩUI: A Distributed Graphical User Interface for the Interactive Proof System ΩMEGA. In *Proceedings of UITP-98*, Eindhoven, The Netherlands, July 13–15 1998.

[11] H. A. Simon. *Models of Bounded Rationality.* MIT Press, Cambridge, 1982.

[12] Gert Smolka. The Oz Programming Model. In J. v. Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer-Verlag, 1995.

[13] S. Zilberstein. Models of Bounded Rationality. In *AAAI Fall Symposium on Rational Agency*, Cambridge, Massachusetts, November 1995.