Requirement-Based Cooperative Theorem Proving

Dirk Fuchs*
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern
Germany
E-mail: dfuchs@informatik.uni-kl.de

Abstract

We examine an approach for demand-driven cooperative theorem proving. We briefly point out the problems arising from the use of common success-driven cooperation methods, and we propose the application of our approach of requirement-based cooperative theorem proving. This approach allows for a better orientation on current needs of provers in comparison with conventional cooperation concepts. We introduce an abstract framework for requirement-based cooperation and describe two instantiations of it: Requirement-based exchange of facts and sub-problem division and transfer via requests. Finally, we report on experimental studies conducted in the areas superposition and unfailing completion.

*The author was supported by the *Deutsche Forschungsgemeinschaft (DFG)*.

1 Introduction

Automated deduction is—at its lowest level—a search problem that spans huge search spaces. In the past, many different calculi have hence been developed in order to cope with problems stemming from the area of automated theorem proving, e.g. the superposition calculus ([BG94]) or certain kinds of tableau calculi (e.g. [Fit96]). Furthermore, the general undecidability of problems connected with (automated) deduction entails an indeterminism in the calculi that has to and can only be tackled with heuristics. Hence, usually a large number of calculi, each of them controllable via various heuristics, can be employed when tackling certain problems of theorem proving.

When studying results of certain theorem proving competitions (e.g., [SS97]) it is recognizable that each calculus or heuristic has its specific strengths and weaknesses. As a matter of fact, for the most domains there is not only one strategy capable of proving all problems of the domain in an acceptable amount of time. Moreover, it is difficult to predict which strategy might be the right one for solving a certain problem. Therefore, a topic that has recently come into the focus of research is the use of different strategies in parallel (see, e.g., [BS97]).

The easiest way to employ different strategies in parallel is to use a system of *competitive* theorem provers (see [Ert92]). Thus, at least the probability is rather high that a well-suited strategy is among them. A better approach, however, is to employ *cooperative* theorem provers. The aim of cooperative theorem proving is to let several provers work in parallel and to exchange information between them. Thus, efficiency should not only be gained by the fact that different strategies work in parallel but also by synergetic effects caused by the exchange of information. Some architectures are proposed for cooperative theorem proving, e.g. in [Sut92, Den95, BH95, Bon96, FD97].

If we take a closer look at existing cooperation approaches we can see that they are in main parts *success-driven* approaches. They are usually characterized as follows: One prover detects a certain information, e.g. a derived fact, that has either been useful for it or that appears—from the prover's point of view—to be useful for a receiving prover in a certain way. Then, this information is transferred to the receiver and integrated into its search state. One main problem regarding this cooperation technique is the *lack of orientation on concrete needs or wishes* of receiving provers. This is mainly due to the fact that on the one hand an information useful for a sender need not necessarily be useful for a receiver. On the other hand, the sender of an information is surely not able to estimate the needs of a receiver as well as the receiver can. Hence, it might be often the case that information is exchanged that is not helpful in order to prove the goal.

Therefore, the aim of this report is to introduce a cooperation model that *orients itself on concrete needs* of theorem provers. Thus, the amount of useless information that is exchanged can be reduced. The main idea of our approach of *requirement-based* theorem proving is not to send information from a sending prover to a receiving prover because the sender believes it may be useful for the receiver, but only to send information as a *respond* to a *request* of the receiving prover that asks for certain kinds of information. Thus, we want to focus on some kind of *demand-driven* cooperation. To

our knowledge, cooperation by exchanging requests and responses has so far only been employed by the resolution-based prover DARES (see [CMM90]). However, requests are in this context only needed to preserve completeness although no prover in the DARES system has all axioms in its set of start clauses: If a prover is not able to perform inferences and the empty clause has not yet been derived it imports via requests facts from other provers that allow it to continue its search. Hence, requests do not introduce a real kind of orientation on concrete needs of provers in the cooperating system. In our approach, however, we utilize requests so as to concentrate on needs of provers in two ways: On the one hand we point out a method for a requirement-based exchange of facts. On the other hand, we will deal with methods to realize problem division and transfer via requests. As we will see, we introduce with the latter an analytic component into provers that do not necessarily work analytically by themselves.

The report is organized in the following way: At first, we introduce basics of automated deduction in section 2 and outline the application domains superposition and unfailing completion. In section 3, we introduce a framework for requirement-based cooperative theorem proving and describe the behavior of our cooperative system. Sections 4 and 5 address concrete aspects of requirements, namely sub-problem transfer via requirements and requirement-based exchange of facts, respectively. After that, we underline the strength of our approach by first empirical studies in section 6. Finally, a discussion and an outlook at possible future work conclude the report.

2 Basics of Automated Deduction

In general, automated theorem proving deals with following problem: Given a set of facts Ax (*axioms*), is a further fact λ_G (*goal*) a logical consequence of the axioms? A fact may be a clause, equation, or a general first or higher-order formula. The definition of "logical consequence" depends heavily on the concrete domain one is interested in.

Commonly, automated theorem provers utilize certain *calculi* for accomplishing the task mentioned above. *Analytic calculi* attempt to recursively break down and transform a goal into sub-goals that can finally be proved immediately with the axioms or with assumptions made during the proof. *Saturation-based calculi* go the other way by continuously producing logic consequences from Ax until a fact covering the goal appears (but also some saturation-based calculi use the goal in inferences). We shall here concentrate on saturation-based calculi.

Typically a saturation-based calculus contains several inference rules of an inference system $\mathcal{I}$ which can be applied to a set of facts (which represents a certain search state). *Expansion* inference rules are able to generate new facts from known ones and add these facts to the search state. *Contraction* inference rules allow for the deletion of facts or replacing facts by other ones, thus contracting the fact base (see, e.g., [Der90]). For sets of facts $\mathcal{M}$ and $\mathcal{N}$, $\mathcal{M} \vdash \mathcal{N}$ denotes that it is possible to derive $\mathcal{N}$ from $\mathcal{M}$ by applying one inference rule.

A common principle to solve proof problems algorithmically with a saturation-based calculus is employed by most systems: Essentially, a theorem prover maintains either implicitly or explicitly a set $\mathcal{F}^P$ of so-called *potential* or *passive facts* from which it

selects and removes one fact λ at a time. After the application of some contraction inference rules on λ, it is put into the set $\mathcal{F}^A$ of *activated facts*, or discarded if it was deleted by a contraction rule (*forward subsumption*). Activated facts are, unlike potential facts, allowed to produce new facts via the application of expanding inference rules. The inferred new facts are put into $\mathcal{F}^P$. We assume the expansion rules to be exhaustively applied on the elements of $\mathcal{F}^A$. Initially, $\mathcal{F}^A = \emptyset$ and $\mathcal{F}^P = Ax$. The indeterministic selection or *activation step* is realized by heuristic means. To this end, a heuristic $\mathcal{H}$ associates a natural number $\mathcal{H}(\lambda) \in \mathbb{N}$ with each $\lambda \in \mathcal{F}^P$. Subsequently, that $\lambda \in \mathcal{F}^P$ with the smallest weight $\mathcal{H}(\lambda)$ is selected. In order to break ties between facts with the same heuristic weight it is possible to use another heuristic. Due to efficiency reasons ties are usually broken according to the FIFO-strategy ("*first in–first out*").

In order to demonstrate the strength of our approach of requirement-based theorem proving we performed experimental studies with the saturation-based calculi *superposition* and *unfailing completion*.

In the area of superposition-based theorem proving we conducted experimental studies with the prover SPASS ([WGR96]). This is an automatic prover for first-order logic with equality. It is based on the superposition calculus (see [BG94]). The inference rules of the superposition calculus can be divided into expansion and contraction rules as we have seen before. The expansion rules contain the common rules of the superposition calculus, i.e. superposition left and right, factoring, equality resolution, and equality factoring. The reduction rules contain well-known rules like subsumption and rewriting. Furthermore, SPASS utilizes additional reduction rules, like the deletion of tautologies and the condensing rule which allows to replace a clause C by $\sigma(C)$ if $\sigma(C) \subset C$.

The *unfailing completion* procedure (see [BDP89]) offers possibilities to develop high performance theorem provers (e.g., DISCOUNT [ADF95]) in pure equational logic. In this context the axioms are always universally quantified equations, the proof goal is an arbitrarily quantified equation. The inference system underlying the unfailing completion procedure is in main parts a restricted version of the superposition calculus. It contains one expansion inference rule—the generation of so-called *critical pairs*—that corresponds to the superposition rule. The contraction rules of unfailing completion correspond to those of the prover SPASS: It is possible to perform rewriting steps, subsume one equation by another, and to delete tautologies.

3 A Framework for Requirement-Based Cooperation

In the following, we want to introduce an abstract framework for requirement-based cooperative theorem proving. We discuss which kinds of requirements may be well-suited for cooperative theorem proving. After that, we describe how requirement-based cooperation can be organized. We introduce a simple architecture for cooperative theorem proving as well as an abstract process model for the cooperative provers. Furthermore, we define the concepts *request* and *response* (to a request) and use them in order to make our abstract process model more concrete.

3.1 Different kinds of requests

The basic idea of requirement-based theorem proving is to establish cooperation between several different theorem provers by exchanging requests and responses to requests. Requests describe certain needs of theorem provers, responses to requests contain information of receivers of requests that may be well-suited in order to fulfill some needs formulated in the requests.

The first problem we have to deal with is to discuss which kinds of requests are sensible, i.e. which kind of information other provers can be asked for. We consider two different types of requests:

Firstly, if it is possible to *divide a proof problem* into various *(sub-) problems*, a prover can require that some of the sub-problems should be solved by other provers. Hence, requests are used for *sub-problem division and transfer*. More exactly, we assume that we are able to divide a proof problem into several *and*-tasks $T_1, \ldots, T_n$ that can be solved independently. Then one can ask other provers to solve some tasks $T_{j_1}, \ldots, T_{j_k}$ $(k < n)$ and tackle only the remaining tasks $\cup_{1 \leq i \leq n} T_i \setminus \cup_{1 \leq z \leq k} T_{j_z}$.

Secondly, it is possible to *demand information* of other provers that may be helpful *for solving the actual proof task*. This kind of request is always sensible, independent of the fact whether all provers tackle the same problem (if a division of problems into sub-problems is not possible or not desirable) or whether provers tackle different problems. In this context requests are no means for (sub-)problem transfer but they are needed so as to cope with the solution of the actual problem. The most profitable information a prover can obtain from others is a set of facts (lemmas) the others have derived. These lemmas can be employed for proving the goal without verifying them again. In particular, a prover should request such facts from others that might be useful for its own proof attempt (regarding certain criteria) but are not already deduced and appear to be difficult to deduce by it (regarding its heuristic). We distinguish between *expansion-based* and *contraction-based requests* for facts: With the first type of request a prover asks for facts that it can use in order to produce descendants which are part of a proof. With the second kind of request a prover asks for facts that are able to contract large parts of its search state.

3.2 An architecture and abstract process model for requirement-based theorem proving

Up to now we have given a rough idea of the kinds of requests we want to deal with. Now, we want to explain in more detail how cooperative theorem proving by exchanging requests and responses could look like. We point out static aspects concerning the architecture of our system of cooperative provers as well as dynamic aspects.

The architecture of our system can be described as follows: On each processor in a network of cooperating computers a theorem prover conducts a search for a proof goal. All provers start with a common original proof goal. Since it is possible, however, that provers divide problems into sub-problems it might be that in later steps of the proof run different provers have different (sub-)goals. We assume that each prover is

assigned a unique number. We either let only different incarnations of the same prover cooperate—differing from each other only in the search-guiding heuristics they employ for traversing the search space—and hence have a network of *homogeneous* provers, or we employ different provers (*heterogeneous* network). We assume that our network of cooperating provers is completely intermeshed, i.e. each prover communicates its requests and responses individually to other provers.

Since proof problems are usually search problems of tremendous difficulty it is important that each prover has enough time to perform inferences independently and to tackle its problem without being permanently interrupted by others. Thus, we decided to let the provers work independently for a while and only cooperate periodically. Basically the working scheme of the provers is characterized by certain phases. While the provers tackle the same problem independently during so-called *working phases* P_w, cooperation takes place during *cooperation phases* P_c. Working phases and cooperation phases alternate with each other. Thus, the sequence of phases is $P_w^0, P_c^0, P_w^1, P_c^1, \ldots$

During the cooperation phases the different theorem provers exchange requests and responses to requests with other provers. Essentially, we can divide the activities during a cooperation phase into four activities of the following *process model:*

1. Determination and transmission of requests to other provers.
2. Transmission of responses to earlier requests of other provers.
3. Receiving and processing foreign requests.
4. Receiving and processing responses to own earlier requests.

As we can recognize, this process model does not allow for an immediate processing of incoming requests, i.e. it is not possible to receive a request, to process it, and to transmit a response in just one cooperation phase. Instead, the response must be transmitted in a later cooperation phase. Thus, the cooperation scheme is somewhat inflexible. Nevertheless, this scheme minimizes the amount of communication because in each cooperation phase a prover must only send/receive one message (containing requests and responses) to/from another prover. Since communication is usually quite expensive we decided to employ this model.

3.3 Fact-represented requests and responses

In order to make the process model more concrete we must at first make our notions of request and response precise. Then we proceed with explaining the four activities of our model. In the following, $\mathcal{F}_{\mathcal{A}}^{A}$ and $\mathcal{F}_{\mathcal{A}}^{P}$ denote the sets of active and passive facts of a prover $\mathcal{A}$.

Definition 3.1 (request)
A request from a prover $\mathcal{A}$ to a prover $\mathcal{B}$ is a quadruple $req = (id_{req}, \lambda_{req}, S_{req}, t_{req})$. $id_{req} \in \mathbb{N}$ is the number of the request, λ_{req} is a fact, S_{req} is a predicate defined on $(\lambda_{req}, \mathcal{F}_{\mathcal{B}}^{A}, \mathcal{F}_{\mathcal{B}}^{P})$, and $t_{req} \in \mathbb{N}$ a time index.

The component id_{req} of a request should be—from the point of view of the sender of the request—a unique number which is needed in order to identify requests and responses (see below).
Each request is represented by a certain *request fact*, the component λ_{req}. I.e. we do not utilize a complex language in order to express needs of a prover, but they are described by the request fact λ_{req} and the predicate S_{req}. The fact λ_{req} represents the request, the predicate S_{req} is a *satisfiability condition* of the request req: If the predicate S_{req} is true the request is completely processed and can hence be answered by the receiver. Now, we show for both kinds of requests, sub-problem transfer and requests that ask for facts, how they can be represented by λ_{req} and S_{req}:
Firstly, if we want to transfer a sub-goal g by a request req we set $\lambda_{req} = g$. The satisfiability condition S_{req} is defined by $S_{req}(\lambda_{req}, \mathcal{F}^A_{\mathcal{B}}, \mathcal{F}^P_{\mathcal{B}})$ iff λ_{req} is proved by the receiver $\mathcal{B}$ to be a logic consequence of its initial axiomatization. Secondly, if we ask for certain facts via a request, λ_{req} is a so-called *schema fact*. This schema fact describes in a certain way how facts look like that may be useful for the sender of a request. In our methods (see section 5), schema facts λ_{req} are valid facts, i.e. logic consequences of Ax. The satisfiability condition S_{req} holds if the receiver $\mathcal{B}$ has at least one fact $\bar{\lambda} \in \mathcal{F}^A_{\mathcal{B}} \cup \mathcal{F}^P_{\mathcal{B}}$ that corresponds to the schema given through λ_{req}. This is tested via a *correspondence predicate* C, i.e. in this case $S_{req}(\lambda_{req}, \mathcal{F}^A_{\mathcal{B}}, \mathcal{F}^P_{\mathcal{B}})$ iff $\exists \bar{\lambda} \in \mathcal{F}^A_{\mathcal{B}} \cup \mathcal{F}^P_{\mathcal{B}} : C(\lambda_{req}, \bar{\lambda})$. Different methods for requesting facts can be developed by using different methods for identifying schema facts and constructing correspondence predicates. We present two different ways in sections 5.1 and 5.2.
In order to send requests to other provers it is necessary to employ a description method for the satisfiability conditions. An easy method could be, e.g., to describe different conditions through different key words. At least it is necessary that all cooperating provers know the key words so as to understand requests of other provers.
The time index t_{req} is the maximal number of working phases which are allowed to take place between the receipt of the request and the transmission of its response. The idea behind the use of such a time index is that the receiver of the request should not work mainly on the request but on its own proof attempt. Requests of other provers should be tackled besides the provers own activities. Since we do not want to put too much load on each prover through requests of others we restrict the processing of requests to a fixed duration. Note that we also use the time index t_{req} for defining the predicate S_{req}. We add the condition "time limit t_{req} is exceeded" as a conjunctive condition to S_{req} when dealing with requests for facts. Thus, we achieve that all facts which fulfill the correspondence predicate S_{req} after the expiration of the time limit are inserted into the response set (see below).
Responses are in analogy to requests represented by facts:

Definition 3.2 (response)
A response of a prover $\mathcal{A}$ to a prover $\mathcal{B}$ is a triple $rsp = (id_{rsp}, B_{rsp}, \Lambda_{rsp})$.
$id_{rsp} \in \mathbb{N}$ is the number of the response, B_{rsp} is a Boolean value, and Λ_{rsp} a set of facts.

The component id_{rsp} of a response equals the number of the respective request that is being answered. The Boolean value B_{rsp} indicates whether or not the responder could

process the request successfully (regarding the satisfiability condition S_{req}) within the time limit given by t_{req}. The *response set* Λ_{rsp} is a set of facts which represents the answer to a request. If we respond to a request that transferred a sub-problem usually $\Lambda_{rsp} = \emptyset$. In this context it is only interesting whether the sub-problem could be proved, i.e. we can ignore the response set. If a prover responds to a request $(id_{req}, \lambda_{req}, S_{req}, t_{req})$ for facts, S_{req} is based on the correspondence predicate C, and I is the set of axioms, Λ_{rsp} is given by

$$\Lambda_{rsp} = \begin{cases} \emptyset & , B_{rsp} \equiv false \\ \{F_1, \ldots, F_{max_{rsp}} : (\forall i : (I \models F_i \wedge C(\lambda_{req}, F_i)))\} & , \text{ otherwise} \end{cases}$$

Since the response set is a logic consequence of the initial system of facts (which is common for all provers), the receiver of facts is allowed to integrate them into its own search state.

By employing these definitions we are able to outline the four different activities of the process model in more detail:

The determination and transmission of requests is performed as follows: At first it is necessary to identify on the one hand sub-problems that should be tackled by other provers, on the other hand schemata of such facts that appear to be useful for proving the goal but are not in the current system of facts of the prover (see section 5 for details). After that, a unique id_{req} as well as a suitable time index t_{req} is assigned to each sub-problem or schema fact that should be sent to another prover via a request req. The satisfiability condition S_{req} must—as mentioned above—be described by a suitable key word. The next step is to insert each request into a queue Req of open requests of the sender and to transmit the request to other provers which are part of the network. More exactly, we transmit all requests that ask for facts to every prover which is part of the network, but transmit each sub-problem only to one other prover.

In order to receive and respond to requests of other provers it is necessary to have also queues Req^i of open requests of other provers i. In order to respond to such requests, requests $req \in Req^i$ must be checked. If req is fulfilled a suitable response can be transmitted and req can be deleted from the queue Req^i. Otherwise, it is necessary to check whether the time limit has been exceeded. If this is true, the response $(id_{req}, false, \emptyset)$ must be communicated to the sender of the request and req must be deleted from the queue. In general, if a request is deleted from a request queue it is necessary to delete all possible offspring of this request. We will deal with this topic in more detail in sections 4 and 5.

If a prover receives a request req from another prover i, firstly req is inserted into Req^i. Secondly, the processing of the request is initiated. E.g., a request concerning the solution of a new sub-problem entails that the new sub-goal must be integrated into the search state of the receiver. Again, we deal with this very issue in more detail in sections 4 and 5 because the processing depends heavily on the concrete type of the request.

When receiving a response rsp to a request it is at first necessary to determine the original request $req \in Req$ with the help of the id_{rsp} component of the response. If the

request has been processed successfully one can—if a sub-problem was transmitted by the request—consider the respective sub-problem to be solved or—if facts have been asked for—integrate the response set Λ_{rsp} into the search state. If the request has not been processed successfully the sender can use this information in future. E.g. it might be sensible to utilize another division of the problem into sub-problems or to ask for different facts in future. Finally, req has to be deleted from Req.

4 Sub-problem Transfer by Requirements

In this section we want to present a method for transferring sub-problems via requests. We restrict ourselves to the area of first-order theorem proving with the superposition calculus, i.e. henceforth facts are first-order clauses which may contain equality. We use the following notation: Literals are usually written in lower case like l, clauses in upper case like $C, D, \ldots$ Calligraphic letters like $\mathcal{M}, \mathcal{N}, \ldots$ usually denote sets of clauses. Moreover, for literals we define $\sim l$ by $\sim l = l'$, if $l \equiv \neg l'$, and $\sim l = \neg l$, otherwise. For a clause $C = \{l_1, \ldots, l_n\}$, $\sim C$ is the set of clauses $\sim C = \{\{\sim l_1\}, \ldots, \{\sim l_n\}\}$. If C is a clause $V(C)$ denotes the set of different variables in C.

In order to realize requirement-based cooperation by transferring sub-problems we employ our abstract model from section 3. It is only needed to make some aspects more precise that we could not describe in the preceding section due to the generality of the model: Firstly, we have to answer the question how we can identify certain request clauses, that is, how we can identify certain sub-problems. Secondly, we must introduce techniques for managing our different sub-problems (so as to interpret responses to requests correctly, see below). Finally, we have to develop a method well-suited for processing sub-problems of other provers without neglecting the own proof attempt.

4.1 Identifying sub-problems

We start with the identification of sub-problems: In the following, we assume that our proof problem is given as a set $\mathcal{M}$ of clauses whose inconsistency is to be proved (by deriving the empty clause). Note that this is not a restriction in comparison with our original notion of a proof problem because $Ax \models C$ iff $Ax \cup \sim C$ is inconsistent. In the following, we will call such clauses from $\sim C$ *goal clauses*. Now, consider this situation:

Definition 4.1 (i-AND-partition)
Let $\mathcal{M} = \mathcal{M}' \cup \{C\}$, $\mathcal{M}'$ be a set of clauses, C be a clause. We call $(P_i)_{1 \leq i \leq n}$ an i-AND-partition of C regarding $\mathcal{M}$ iff

- $(P_i)_{1 \leq i \leq n}$ is a partition of C, i.e. $\cup_{1 \leq i \leq n} P_i = C$, $P_i \cap P_j = \emptyset$ if $i \neq j$, and
- $\mathcal{M}$ is inconsistent iff $\forall i, 1 \leq i \leq n : \mathcal{M}' \cup \{P_i\}$ is inconsistent.

In the case that the inconsistency of a set $\mathcal{M}$ should be proved and we have identified an i-AND-partition of a clause $C \in \mathcal{M}$, we have also identified a division of our original proof problem into n sub-problems $\wp_i =$ "$\mathcal{M}' \cup \{P_i\}$ is inconsistent".

Such an approach for dividing a problem into sub-problems is viable because there is an easy method for identifying i-AND-partitions of a clause:

Example 4.1 Let $\mathcal{M} = \mathcal{M}' \cup \{\{P(a), Q(y)\}\}$. Then, the literals in $C = \{P(a), Q(y)\}$ do not share common variables, i.e. in order to show the inconsistency of $\mathcal{M}$ it is sufficient to prove the inconsistency of both $\mathcal{M}' \cup \{\{P(a)\}\}$ and $\mathcal{M}' \cup \{\{Q(y)\}\}$ independently. Thus, we achieve on the one hand a division of the original task into two tasks. On the other hand if we have, e.g., shown that $\mathcal{M}' \cup \{\{P(a)\}\}$ is inconsistent, we can employ the lemma $\{\neg P(a)\}$ when trying to prove the inconsistency of $\mathcal{M}' \cup \{\{Q(y)\}\}$.[1]

The above example motivates a division of an original problem into sub-problems as follows:

Theorem 4.1 *Let $\mathcal{M}$ be a set of clauses, $\mathcal{M} = \mathcal{M}' \cup \{C\}$ for a set of clauses $\mathcal{M}'$ and a clause C. Let $(P_i)_{1 \leq i \leq n}$ be a partition of C and $V(P_i) \cap V(P_j) = \emptyset$ for $i \neq j$. Moreover, let the sets of clauses $\mathcal{N}_i$ $(1 \leq i \leq n)$ be defined by $\mathcal{N}_i = \mathcal{M}' \cup \{\{\sim l\} : l \in P_j, j < i, V(P_j) = \emptyset\}$. Then it holds:*

1. *$(P_i)_{1 \leq i \leq n}$ is an i-AND-partition of C regarding $\mathcal{M}$.*
2. *$\mathcal{M}$ is inconsistent iff $\forall i, 1 \leq i \leq n : \mathcal{N}_i \cup \{P_i\}$ is inconsistent.*

Proof:

1. Since $(P_i)_{1 \leq i \leq n}$ is a partition of C it is sufficient to show: $\mathcal{M}$ is inconsistent iff $\forall i, 1 \leq i \leq n : \mathcal{M}' \cup \{P_i\}$ is inconsistent.
   If $\mathcal{M}$ is inconsistent then there is a derivation with the superposition calculus $\mathcal{M} \vdash^*_{sup} \square$. It is easy to construct for all i a derivation $\mathcal{M}' \cup \{P_i\} \vdash^*_{sup} \square$ by omitting superposition steps involving literals from $C \setminus P_i$.
   If $\forall i, 1 \leq i \leq n : \mathcal{M}' \cup \{P_i\}$ is inconsistent, there are n derivations $\mathcal{M}' \cup \{P_i\} \vdash \ldots \vdash \square$. Because of the fact that $V(P_i) \cap V(P_j) = \emptyset$ for $i \neq j$, we can construct following derivation: $\mathcal{M} = \mathcal{M}' \cup \{C\} = \mathcal{M}' \cup \bigcup_{1 \leq i \leq n} \{P_i\} \vdash \ldots \vdash \mathcal{M}^{(2)} \cup \bigcup_{2 \leq i \leq n} \{P_i\} \vdash \ldots \vdash \mathcal{M}^{(n)} \cup \{P_n\} \vdash \ldots \vdash \square; \forall k, 2 \leq k \leq n : \mathcal{M}' \subseteq \mathcal{M}^{(k)}$. Hence, $\mathcal{M}$ is inconsistent.

2. If $\mathcal{M}$ is inconsistent we can infer because of part 1 that $\forall i, 1 \leq i \leq n : \mathcal{M}' \cup \{P_i\}$ is inconsistent. Since $\forall i, 1 \leq i \leq n : \mathcal{M}' \subseteq \mathcal{N}_i$ it is clear that also $\forall i, 1 \leq i \leq n : \mathcal{N}_i \cup \{P_i\}$ is inconsistent.
   Now, let $\forall i, 1 \leq i \leq n : \mathcal{N}_i \cup \{P_i\}$ be inconsistent. We show inductively: $\forall i, 1 \leq i \leq n : \mathcal{N}_i \cup \{P_i\}$ is inconsistent $\rightsquigarrow \forall i, 1 \leq i \leq n : \mathcal{M}' \cup \{P_i\}$ is inconsistent. Then, the inconsistency from $\mathcal{M}$ is a consequence of part 1 of the theorem.
   $n = 1$: This is trivial because $\mathcal{N}_1 = \mathcal{M}'$.
   $n > 1$: Since through the induction hypotheses it holds $\mathcal{M}' \cup \{P_j\}$ is inconsistent for $j < n$: $\mathcal{M}' \models \{\{\sim l\} : l \in P_j, j < n, V(P_j) = \emptyset\} =: \Delta$. Because of the fact that $\mathcal{N}_n = \mathcal{M}' \cup \Delta$ it is obvious that if $\mathcal{N}_n \cup \{P_n\}$ is inconsistent also $\mathcal{M}' \cup \{P_n\}$ is inconsistent. $\square$

[1] Note that the clause {P(a)} is ground.

The theorem points out a method for creating new sub-problems that we can employ in the following manner: On each processing node—*if we have not already divided the problem into sub-problems*—we check for each activated clause C which is a descendant of a goal clause whether it can be partitioned into $(P_i)_{1 \leq i \leq n}$, $n \geq 2$, $V(P_i) \cap V(P_j) = \emptyset$ for $i \neq j$. If m is the number of cooperating provers we limit the value n by $1 < n \leq m$. Then, each prover that is able to find such a partition of a clause C distributes $n-1$ tasks $\wp_2, \ldots, \wp_n$ to other provers via requests (a prover obtains exactly one of these sub-problems) and tackles the remaining task $\wp_1$. That is, it replaces the clause C with its sub-clause P_1. The tasks which are sent to other provers are stored in a list $R = (\wp_2, \ldots, \wp_n)$. Note that the time index t_{req} of each request should be chosen in such a way that the receivers have enough time for solving the problem. In our experiments we have chosen the value $t_{req} = 10$. Nevertheless, the prover must possibly tackle also the remaining sub-problems. This is because there is no guarantee that the other provers can give positive answers to the requests within their time limits.

This kind of problem transfer has two advantages: On the one hand, a theorem prover that has divided the problem can work more efficiently. This is due to the fact that it works with smaller clauses. Hence, inferences can be performed very quickly and the branching-rate of the search does not increase so much. On the other hand, it is possible to obtain interesting lemmas from other provers if they are able to give positive responses to requests. All in all, in a way our kind of problem division and transfer introduces some kind of analytic component into saturation-based theorem proving.

4.2 Managing and processing sub-problems

The main problem which is caused by this kind of problem division and transfer is that both sender and receiver of a request have to work with clauses that are in general not logic consequences of the initial set of clauses. This is because a sub-clause of a clause need not logically follow from the clause. Thus, we must develop mechanisms so as to work with such clauses.

We start with the *sender* of a request and assume that it tackles the sub-problem $\wp_i =$ "$\mathcal{M}' \cup \{P_i\}$ is inconsistent" on its own by adding P_i to its search state and deleting C. We examine two aspects: On the one hand, we describe how the inference process can and should be organized so as to cope with clauses which are not semantically valid. On the other hand, we describe the necessary actions that have to be conducted when an empty clause has been derived or a respond to a request has been received from another prover.

In order to work with a semantically invalid clause P_i during the inference process the sender introduces a *tag* for P_i and all descendants of P_i. This tag is simply the number of the prover. It indicates that these clauses are no logic consequences of the initial clause set but descendants of semantically invalid clauses. The practical realization is as follows: We do not work any longer with clauses C but with clauses with tag (C, τ). Either $\tau = \epsilon$ or $\tau = n \in \mathbb{N}$. $\tau = \epsilon$ denotes that the clause C is untagged, i.e. it is a logic consequence of the initial set of clauses. If $\tau = n$, n being the number of the sender, then the clause C is a descendant of P_i. In order to perform inferences we replace the inference system $\mathcal{I}$ that each prover employs by inference system $\mathcal{I}^\tau$:

Definition 4.2 (Inference system $\mathcal{I}^\tau$)
Let $\mathcal{I}$ be an inference system which works on sets of clauses. Then we construe the inference system $\mathcal{I}^\tau$ working on sets of tagged clauses as follows:

1. For each expanding inference rule
   $\{C_1, \ldots, C_n\} \vdash \{C_1, \ldots, C_n, C\}; \mathit{Cond}(C_1, \ldots, C_n)$
   in $\mathcal{I}$, $\mathcal{I}^\tau$ contains the rules
   $\{(C_1, \epsilon), \ldots, (C_n, \epsilon)\} \vdash \{(C_1, \epsilon), \ldots, (C_n, \epsilon), (C, \epsilon)\}; \mathit{Cond}(C_1, \ldots, C_n)$ and
   $\{(C_1, \tau_1), \ldots, (C_n, \tau_n)\} \vdash \{(C_1, \tau_1), \ldots, (C_n, \tau_n), (C, \tau)\}; \mathit{Cond}(C_1, \ldots, C_n) \wedge$
   $\exists k \in \mathbb{N} : (\exists i : \tau_i = k \wedge \forall i : \tau_i \in \{k, \epsilon\} \wedge \tau = k)$

2. For each contracting inference rule
   $\{C_1, \ldots, C_n, C\} \vdash \{C_1, \ldots, C_n, C'\}; \mathit{Cond}(C_1, \ldots, C_n, C)$
   in $\mathcal{I}$ (C might be deleted by the inference), $\mathcal{I}^\tau$ contains the rules
   $\{(C_1, \epsilon), \ldots, (C_n, \epsilon), (C, \tau\} \vdash \{(C_1, \epsilon), \ldots, (C_n, \epsilon), (C', \tau)\}$;
   $\mathit{Cond}(C_1, \ldots, C_n, C)$ and
   $\{(C_1, \tau_1), \ldots, (C_n, \tau_n), (C, \tau)\} \vdash \{(C_1, \tau_1), \ldots, (C_n, \tau_n), (C, \tau), (C', \tau')\}$;
   $\mathit{Cond}(C_1, \ldots, C_n, C) \wedge \exists k \in \mathbb{N} : (\exists i : \tau_i = k \wedge \forall i : \tau_i \in \{k, \epsilon\} \wedge \tau \in \{k, \epsilon\} \wedge \tau' = k)$

Hence, expansion inferences are performed in such a way that a clause C which is a result of an expanding inference with premises $C_1, \ldots, C_n$ is tagged, if some clauses C_i are tagged. Untagged clauses can contract every other clause and in that case the tag remains unchanged. If tagged clauses are able to contract an untagged clause it is necessary to store a copy of the (un-contracted) untagged clause. Otherwise, completeness may be lost if the processing of the request is finished and its offspring has been eliminated. Such copies are stored in a list $\mathcal{D}_j$, j being the number of the prover.

Now, if a prover j is able to derive the empty clause $\square$ the tag of the clause is checked. If the clause is untagged a proof of the original goal has been found. If it is tagged with the number of the prover, the current sub-problem has been solved. In the latter case the following activities take place: All clauses which are tagged with the prover's number j are deleted and the clauses $D \in \mathcal{D}_j$ are integrated untagged into the search state. If the list of open sub-problems $R = ()$ all sub-problems have been solved, i.e. also the original problem. Otherwise, a new sub-problem is chosen from R and processed as described. Note that we do not require to choose a sub-problem that is not distributed to other provers because there is no guarantee that these problems can be solved by others within the time limit.

This modified inference scheme allows us also to process incoming responses easily: If a sub-problem $\wp_j$ has been solved by another prover it must only be eliminated from R. Moreover, if the request clause P_j was ground the clauses being elements of $\{\{\sim l\} : l \in P_j\}$ can be utilized as new lemmas in future.

When working as a *receiver* of open sub-problems we proceed in a similar way: Each sub-goal received from another prover is tagged with the number of this prover and is added to the search state. Inferences between tagged and untagged clauses are performed as described, i.e. we employ inference system $\mathcal{I}^\tau$. Thus, we forbid inferences between clauses having different tags in order to avoid inconsistency.

If an empty clause is derived by a prover, the activities are as follows: If the empty clause is untagged or tagged with the prover's number the activities are as described above. If it is tagged with the number of another prover i all clauses with this tag are deleted and the clauses from $\mathcal{D}_i$ are added to the search state. Furthermore, the sub-problem is considered to be solved, i.e. a positive response can be sent to the sender of the request in the next cooperation phase.

In principle, the division of a problem into sub-problems can be conducted recursively, i.e. it is possible to divide sub-problems into new sub-sub-problems and so on. Hence, we need a more complex tree-style management of sub-problems. Because of the fact that this kind of recursive sub-problem division follows the principles of the simple division that we have described (with more complicated tag mechanisms) we are not going to explain it in more detail. For our experimental studies (see section 6) we employed a recursive sub-problem division and limited the depth of the resulting sub-goal tree to the value 3.

5 Requirement-Based Exchange of Facts

In this section, we present two different methods for exchanging facts via requests and responses. Note that we restrict ourselves again to the area of first-order theorem proving with equality, i.e. facts correspond in the following to first-order clauses. We assume that all provers employ the superposition calculus and additional contraction rules like subsumption and rewriting.

The principle scheme of a requirement-based exchange of clauses is already known through the abstract model from section 3. However, it is necessary to describe two remaining aspects: Firstly, we must introduce methods for detecting clauses other provers should be asked for. I.e. we have to determine request clauses (schema clauses) $\mathcal{R}_i = \{C^i_1, \ldots, C^i_{max_{req}}\}$ in each cooperation phase P^i_c. As already mentioned, these request clauses are sent to each other prover in the network via max_{req} requests. Secondly, we have to deal with the issue of how such requests can be processed by the receivers, i.e. we have to make precise how to compute a response set $\mathcal{C}_{rsp}$ of a response rsp to a request req.

The basic idea of requests for clauses is that a theorem prover tries to get those clauses from other provers that appear to be part of a proof, but are not already derived and seem to be difficult to derive. The main problem in this context is that—because of the general undecidability of first-order theorem proving—it is impossible to predict whether or not a clause is part of a proof. Thus, there is no criterion for identifying clauses that a prover should ask for. However, a prover is able to estimate whether some of its own already activated clauses possibly contribute to a proof. E.g., if we want to conclude a proof by deriving the empty clause $\square$ (which does not contain some literals) it is more likely that short clauses, i.e. clauses with few literals, contribute to a proof than clauses with a lot of literals. Then, if we assume that a prover has identified a set $\mathcal{M}$ of "interesting" activated clauses, interesting clauses other provers can be asked for are such clauses that allow for producing descendants with clauses from $\mathcal{M}$. Perhaps some of this offspring can contribute to a proof. In conclusion, we

can say that a prover should request such clauses from other provers that enable it to perform expanding inferences with these and own interesting clauses. Thus, we call such requests *expansion-based requests.*

There is also another concept for a requirement-based exchange of clauses: Indeed it is difficult to predict whether or not a clause *contributes to a proof* but nevertheless it is possible to recognize whether a clause is *useful for the search for the proof.* If a clause is able to contract many other clauses it is definitely useful for the search process because it helps to save both memory and computation effort. Thus, it is also interesting to require that other provers should send clauses that allow for a lot of contracting inferences. We call these requests *contraction-based requests.*

In the following, we examine for both kinds of requests how they can be determined by the sender and processed by the receiver.

5.1 Expansion-based requests

Determining request clauses: At first, we deal with the determination of request clauses. When employing expansion-based requests, in each cooperation phase P_c^i a prover determines request clauses $\mathcal{R}_i = \{C_1^i, \ldots, C_{max_{req}}^i\} \subset \mathcal{F}^A$. Each request clause should be untagged, i.e. a valid clause which could be derived by the sender of the request when only employing the initial set of clauses. With these request clauses other provers should be asked for clauses that are able to produce descendants with some of the clauses from $\mathcal{R}_i$. The clauses C_j^i $(1 \leq j \leq max_{req})$ should be the clauses of the prover that appear to be most likely to contribute to a proof. More exactly, the clauses $C_1^i, \ldots, C_{max_{req}}^i$ should be optimal regarding a judgment function φ. This function φ rates the probability that a clause is part of a proof.

The realization of the judgment function φ is the most crucial point with regard to the performance of a system based on expansion-based requests. Hence, we want to deal with the realization of φ in some more detail: If a clause C should be judged that does not contain equations as literals we could e.g. use the following technique: Since it is the aim of a prover to derive the empty clause a clause is considered to be the better the less literals it has. Thus, we could use the formula $\varphi(C) = \frac{1}{|C|}$. We adopted and refined the method as follows: In addition to the length of a clause we take into account that clauses having literals with a rather "flat" syntactic structure can often be used for expansion inference steps. This means that these literals do not have deep sub-terms that prevent them from taking part in unification which is the essential operation for expansion inference rules like resolution. Thus, considering the number of literals and the syntactic structure of the literals we obtain the following weighting function:

Definition 5.1 (weighting function φ for expansion-based requests)
The weighting function φ for expansion-based requests is defined on clauses by

$$\varphi(C) = \sum_{i=1}^{n} \varphi_{Lit}(l_i); C = \{l_1, \ldots, l_n\}$$

The function φ_{Lit} is defined as follows: $\varphi_{Lit}(l) = -\varphi^H_{Lit}(l,0)$, if l is positive, and $\varphi_{Lit}(l) = -\varphi^H_{Lit}(l',0)$, if $l \equiv \neg l'$. The function φ^H_{Lit} can be computed by

$$\varphi^H_{Lit}(l,d) = \begin{cases} 1+d & ;l \text{ is a variable} \\ 2+d+\sum_{i=1}^{n} \varphi^H_{Lit}(t_i, d+1) & ;l \equiv f(t_1,\ldots,t_n) \text{ or } l \equiv P(t_1,\ldots,t_n), f \text{ is a function symbol}, P \text{ a predicate symbol} \end{cases}$$

φ judges a clause the better the less literals it has, and the less symbols and deep sub-terms each literal has. Thus, the function complies with our demands formulated above.

If equality is involved, i.e. we have equations as literals, it is sensible to refine φ_{Lit}: If an inequation $s \neq t$ is given that can be used to derive $s' \neq s'$ within few inferences we have found the empty clause since it can be derived by $s' \neq s'$ and the reflexivity of the equation symbol. Hence, if we have an inequation with nearly identical left and right hand sides we can possibly use the equation so as to derive the empty clause. Therefore, φ_{Lit} should rate inequations the better the less different the left and the right hand sides are. In order to measure such differences we employ a method used in [Fuc97] in order to measure differences. If an equation $s = t$ is to be judged we employ again φ_{Lit} from the preceding definition if no goal clause was an equation. Otherwise, we measure some kind of similarity between $s = t$ and the goal equation(s). E.g. it is sensible to check whether it is possible to derive the goal by applying superposition to $s = t$. We used the similarity measures as described in [DF94] in order to realize φ_{Lit}.

Computing response sets: The second main aspect—besides the determination of request clauses—is the processing of requests by their receivers. Essentially, we have to deal with the problem of computing a response set $\mathcal{C}_{rsp}$ regarding a request req. In the following, we employ the sets $Inf(\mathcal{M}, C) = \{C' : C' \text{ is derivable with one expanding inference step involving } C \text{ and some clauses from } \mathcal{M}\}$ and $Inf^*(\mathcal{M}, C) = \{C' : C' \text{ is derivable via some inferences from } C \text{ and clauses from } \mathcal{M}\}$ in order to describe the processing of request clauses. Furthermore, let $\mathcal{F}$ be the set of active and passive clauses of the receiver of a request req containing the request clause C_{req}. As already informally described, the easiest method for determining a response set $\mathcal{C}_{rsp}$ is to insert such clauses into $\mathcal{C}_{rsp}$ that allow for expanding inferences with C_{req}. If we employ our notion from section 3, i.e. $\mathcal{C}_{rsp} \subseteq \{\bar{C} : \bar{C} \in \mathcal{F} \wedge C(C_{req}, \bar{C})\}$, the correspondence predicate is defined by $C(C_{req}, \bar{C})$ iff $Inf(\{\bar{C}\}, C_{req}) \neq \emptyset \wedge (\bar{C}$ is a logic consequence of the initial clauses). A disadvantage of this approach is that certain inferences must be performed twice: On the one hand it is necessary to perform expanding inferences with C_{req} at the receiver site in order to determine clauses $\bar{C} \in \mathcal{F}$ which can be involved in expanding inferences together with C_{req}. On the other hand, the receiver of the response set $\mathcal{C}_{rsp}$ must perform exactly the same inferences with C_{req} when it integrates clauses which are elements of $\mathcal{C}_{rsp}$ into its search state. Thus, our refinement of this simple method is as follows: The main idea is to already perform inferences with C_{req} at the responder site and to transmit only such clauses to the sender of the request that are already descendants of C_{req} and some of the clauses of the responder. Thus, the response set is given by $\mathcal{C}_{rsp} = \beta(\{\bar{C} : \bar{C} \in \mathcal{F} \wedge C(C_{req}, \bar{C})\})$, $C(C_{req}, \bar{C})$ iff $\bar{C} \in Inf^*(\mathcal{F}, C_{req}) \wedge (\bar{C}$ is a logic consequence of the initial set of clauses). The function β is responsible

for selecting some of the descendants, i.e. $\beta(Inf^*(\mathcal{F}, C_{req})) \subseteq Inf^*(\mathcal{F}, C_{req})$. We realize β in such a way that we choose max_{rsp} clauses that have maximal weights regarding φ.

$Inf^*(\mathcal{F}, C_{req})$ can be computed either independently from the "normal" inferences after the receipt of a request, or simultaneously to the inferences necessary to tackle the proof problem. We chose the latter approach by integrating C_{req} as an active clause into the search state of the receiver and tagging it with both the number of the sender of the request and the id_{req} of the request req. Hence, we can distinguish C_{req} and its descendants from offspring of other requests. Expansion and contraction inferences involving tagged and untagged clauses are performed as already described in the area of sub-goal transfer via requests, i.e. we employ inference system $\mathcal{I}^\tau$ extended with the possibility to have pairs of natural numbers as tags. But in contrast to before, we forbid contracting inferences if the contracting clause is offspring of a request clause. Hence, we need not introduce a separate list $\mathcal{D}_{C_{req}}$ for each request clause C_{req}. This is sensible because we are almost interested in descendants of C_{req} produced by expansion inferences. Note that if we are able to derive an empty clause which is tagged with the number of an expansion-based request the whole proof problem is solved because all clauses with this tag are logical consequences of the initial clause set. If we are not yet able to derive $\square$ and the time limit t_{req} for the response expires we select in the cooperation phase a set $\mathcal{C}_{rsp}$ via β as described, send a response message, and delete the offspring of the request. Note that the time limit t_{req} should not be too small so as to allow the prover to derive some descendants of a request clause C_{req}.

5.2 Contraction-based requests

By contraction-based requests a theorem prover asks other provers for clauses that are possibly well-suited for contracting, i.e. in our context rewriting and subsuming, many clauses of its clause set $\mathcal{F}$. We deal in the following with the topic of how request clauses can be identified and we point out a method for computing a response set $\mathcal{C}_{rsp}$ regarding a request req with request clause C_{req}.

Determining request clauses: Especially well-suited for reducing the amount of data and computation are clauses that subsume or rewrite clauses that tend to produce much offspring. Hence, other provers should be asked for such clauses. Thus, the set of clauses $\mathcal{M} = \{C : C$ is an active positive unit; C is among the max_{req} largest generators of clauses$\}$ is determined as a set of request clauses in each cooperation phase. This set offers each receiver of the request clauses the possibility to determine clauses which are able to subsume or rewrite them. These clauses are then especially useful for the search for the proof because they can contract clauses from $\mathcal{M}$ which cause much overhead. Note that we restrict ourselves to positive units mainly due to efficiency reasons. The number of clauses which are generated by using a certain clause can simply be counted during the inference process. Hence, we can determine the set $\mathcal{M}$ effectively.

Computing response sets: In order to determine a response set $\mathcal{C}_{rsp}$ regarding a request with request clause C_{req} we insert on the one hand clauses into $\mathcal{C}_{rsp}$ which are

able to subsume C_{req}, on the other hand clauses which are able to rewrite C_{req}. Hence, we have $\mathcal{C}_{rsp} = \mathcal{C}_{rsp,sub} \cup \mathcal{C}_{rsp,rew}$.
If $\mathcal{F}^{A,v}$ contains all active clauses of the responder that are logic consequences of the initial set of clauses, the set $\mathcal{C}_{rsp,sub}$ regarding a request clause C_{req} is simply given by

$$\mathcal{C}_{rsp,sub} = \{\bar{C} : (\bar{C} \in \mathcal{F}^{A,v}, \exists\sigma : \sigma(\bar{C}) \equiv C_{req})\}$$

We set $C_1(C_{req}, \bar{C})$ iff $\bar{C} \in \mathcal{F}^{A,v} \wedge \exists\sigma : \sigma(\bar{C}) \equiv C_{req}$. Determining a set $\mathcal{C}_{rsp,rew}$ of clauses which are able to rewrite C_{req} is more complicated as before because we must consider the ordering $\succ$ each prover uses for performing inferences. We restrict ourselves in the following to response sets containing only positive equations because only rewriting with such clauses contracts a clause without simultaneously introducing new literals. Since we cannot rewrite with the minimal side of an equation (regarding $\succ$), we must at first identify the sides relevant for rewriting and transform clauses $\bar{C} \in \mathcal{F}^{A,v}$ with following function θ to sets $\theta(\bar{C})$: If the sender of the request and the responder have an identical ordering $\succ$, we utilize

$$\theta(\bar{C}) = \begin{cases} \{s\} & ;\bar{C} \equiv s \doteq t, s \succ t \\ \{s,t\} & ;\bar{C} \equiv s = t, s \not\succ t, t \not\succ s \end{cases}$$

Hence, we consider only the left hand side of a rewrite rule but both sides of an equation. Otherwise, if sender and receiver employ different orderings, we employ

$$\theta(\bar{C}) = \{s,t\}; \bar{C} \equiv s = t$$

Then, it is necessary to check whether terms from $\theta(\bar{C})$ match to a sub-term of C_{req}. Such clauses can be inserted into $\mathcal{C}_{rsp,rew}$ and send via respond messages.
In the following, $O(C_{req})$ denotes the set of positions in the request clause C_{req} and $C_{req}|p$ the sub-term of C_{req} at position p. Then, we obtain:

$$\mathcal{C}_{rsp,rew} = \{\bar{C} : (\bar{C} \in \mathcal{F}^{A,v}, \exists(\sigma, \bar{C}' \in \theta(\bar{C}), p \in O(C_{req})) : \sigma(\bar{C}') \equiv C_{req}|p)\}$$

We set $C_2(C_{req}, \bar{C})$ iff $(\bar{C} \in \mathcal{F}^{A,v}, \exists(\sigma, \bar{C}' \in \theta(\bar{C}), p \in O(C_{req})) : \sigma(\bar{C}') \equiv C_{req}|p)$. The correspondence predicate is then given by $C(C_{req}, \bar{C})$ iff $C_1(C_{req}, \bar{C}) \vee C_2(C_{req}, \bar{C})$. The time limit t_{req} of a contraction-based request req should be chosen quite small because the responder need not perform inferences in order to determine a response set $\mathcal{C}_{rsp}$ but only has to check its active clauses $\mathcal{F}^A$. We employed hence the minimal time limit $t_{req} = 1$.

6 Experimental Results

In order to examine the potential of our cooperation concepts we conducted our experimental studies in the light of different domains of the problem library TPTP (see [SSY94]). As we have already mentioned, we restrict ourselves to the area of

superposition-based theorem proving and couple the provers SPASS—which employs the superposition calculus and additional reduction rules like rewriting—and DISCOUNT. DISCOUNT is a prover for pure equational logic which utilizes the unfailing completion procedure which can be seen as a restricted version of the superposition calculus with additional rewrite rules. Note that we coupled provers that are already quite powerful. Hence speed-ups w.r.t. the sequential provers are not due to their inefficiency. Our test set consisted of pure unit equality problems as well as problems specified in full first-order logic with equality. Thus, we can reveal that our cooperation concept achieves cooperation among different provers in an area where both provers are complete as well as in an area where one prover is only able to support the other but not to solve the original problem. Hence, we show that our concept is well-suited for provers having equal rights as well as for provers being in a master-slave relation. At first, we describe our experimental settings for both areas. After that, we present an excerpt of the experimental results.

6.1 Test setting

Since both calculi—superposition and unfailing completion—are complete for pure equational logic (EQ), SPASS and DISCOUNT can work as partners having equal rights for problems of EQ. Thus, we let each prover send requests and responses to requests to its counterpart. However, not all kinds of requests can be employed when dealing with problems specified in equational logic. Because of the fact that $|C| = 1$ for all clauses C it is not possible to divide a problem into sub-problems as described in section 4. Thus, we must omit requests dealing with sub-problem transfer. Nevertheless, expansion- and contraction-based requests for exchanging clauses can be utilized. We exchanged expansion-based requests and responses in the following manner: In each cooperation phase each prover determines $max_{req} = 10$ request clauses to be distributed to the other prover. In order to respond to an expansion-based request we inserted $max_{rsp} = 3$ clauses into the respective response set $\mathcal{C}_{rsp}$. As we have already mentioned, it is sensible to give the responder enough time for processing the request. Therefore, we set the time limit $t_{req} = 3$. In order to exchange contraction-based requests we restricted the size of the set M of largest generators of clauses to $max_{req} = 10$. As already stated, the time limit t_{req} was given by $t_{req} = 1$.

In the area of full first-order logic with equality (PL1EQ) DISCOUNT is not able to prove every valid goal because it can only deal with equations. Nevertheless, SPASS and DISCOUNT can work in some kind of master-slave relation because DISCOUNT is at least able to infer many clauses from the part of the search state it can traverse that may be useful for SPASS. We decided to utilize following decomposition of a proof problem for DISCOUNT, represented by a set of clauses $\mathcal{C}$ whose inconsistency should be shown: Each positive equation $P(t_1, \ldots, t_n) = true$ or $s = t$ of $\mathcal{C}$ is chosen as an axiom for DISCOUNT, each negative equation acts as a proof goal. We only considered examples where we could isolate enough positive equations such that the completion of DISCOUNT did not stop. In the case that no negative equation was an element of $\mathcal{C}$ DISCOUNT worked without a proof goal, i.e. in a completion mode. Because of the fact that DISCOUNT cannot prove every valid goal we decided to let only

SPASS send expansion-based requests for clauses. We extended DISCOUNT so as to allow it to perform superposition with its equations and clauses received from SPASS. Contraction-based requests were exchanged by both provers since clauses that allow to save memory and computation time are useful for both provers. We have chosen the same parameter setting as in the area of unit equality. Because of the fact that in first-order logic with equality a clause can have a length greater than 1 we can transfer sub-problems from SPASS to DISCOUNT. It is only possible, however, to transfer a sub-problem to DISCOUNT which is represented by a negative unit.

In general, we let SPASS work with its standard heuristic that simply considers the number of symbols of a clause. DISCOUNT activated clauses with a goal-oriented heuristic as described in [DF94].

6.2 Results

In order to measure the strength of our cooperation concepts we experimented in the light of various problems taken from TPTP. In all of our test domains we only considered problems that none of the provers could solve within 10 seconds (medium and hard problems). As already said, we omitted also problems where the completion of DISCOUNT stopped before the first cooperation phase started. For all examined problems we could observe that at least one variant of our cooperative system (see below) was either better than each of the coupled provers—the runtime was less or the cooperative provers could solve a problem none of the coupled provers could solve when working alone—or we achieved the same result, that is, neither the cooperative system nor one of the coupled provers could cope with the problem. For illustration purposes we present a small excerpt of these experiments in table 1. In order to allow for a better comparison of our different concepts for sending requests for clauses, we performed experiments for both concepts separately. I.e. we either exchanged only expansion-based requests and responses to the requests or contraction-based requests and responses. Requests that transferred sub-problems were—considering the above restrictions—always exchanged. Results are presented in table 1. Problem names can be found in column 1, the results of SPASS when working alone in column 2. Column 3 displays the run times when using DISCOUNT. In general, the entry "–" denotes that the problem could not be solved within 1000 seconds (all runtimes were achieved on one or two SPARCstations 20). Column 4 shows whether the problem is specified in pure equational logic (EQ) or in first-order logic with equality (PL1EQ). Column 5 displays the run time when employing requests for sub-problem transfer and expansion-based requests for clauses (with the mentioned restrictions), column 6 the respective time when exchanging requests for sub-problem transfer and contraction-based requests for clauses. The last column 7 presents which prover could solve the problem in the cooperating runs.

For all problems we can find at most one cooperation method that allows for a gain of efficiency. This gain of efficiency is sometimes low (`LDA011-2`, `HEN010-5`) but in the prevailing number of cases we achieve high speed-ups (e.g., `BOO007-4`, `ROB022-1`, `ROB023-1`). Furthermore, sometimes it is even possible to solve problems through cooperation that are out of reach for both of the coupled provers (`GRP177-2`, `GRP179-1`). If

problem	SPASS	DISCOUNT	EQ/PL1EQ	expans.	contr.	proved by
BOO007-4	403.4	–	EQ	330.7	144.4	DISCOUNT
GRP177-2	–	–	EQ	–	123.8	DISCOUNT
GRP179-1	–	–	EQ	447.0	63.5	DISCOUNT
LCL163-1	10.0	12.0	EQ	8.2	6.2	DISCOUNT
ROB005-1	–	109.6	EQ	36.6	60.6	SPASS
ROB008-1	–	98.8	EQ	13.5	85.9	SPASS
ROB022-1	15.1	–	EQ	2.3	3.9	SPASS
ROB023-1	204.6	–	EQ	47.3	44.6	SPASS
CIV001-1	24.9	–	PL1EQ	13.0	25.3	SPASS
LDA011-2	35.1	–	PL1EQ	40.2	30.4	SPASS
ROB011-1	105.3	–	PL1EQ	110.7	54.9	SPASS
ROB016-1	9.8	–	PL1EQ	4.3	5.8	SPASS
HEN009-5	309.9	–	PL1EQ	370.8	233.9	SPASS
HEN010-5	68.7	–	PL1EQ	62.9	70.3	SPASS
HEN011-5	41.2	–	PL1EQ	29.3	20.1	SPASS
LCL143-1	16.1	–	PL1EQ	12.4	11.3	SPASS

Table 1: Coupling SPASS and DISCOUNT by exchanging requests and responses

we compare the results achieved by expansion-based requests with those of contraction-based requests we can see that contraction-based requests are mostly the better alternative.

We examine the gains of efficiency in more detail and study at first expansion-based requests: On the one hand, it was sometimes the case that the receiver of an expansion-based request could use the clauses to prove the goal by itself (ROB022-1, ROB023-1). Note that clauses sent via expansion-based requests are chosen in such a way that they appear to be contributing to a proof. Hence, it is not surprising that the receiver could sometimes utilize these clauses for proving the goal. This phenomenon occurred especially in the area of unit equality. On the other hand, in many cases offspring of requests that was sent via response messages could be used by the sender of a request for proving the goal. Especially in the area of full first-order logic with equality DISCOUNT could often generate interesting descendants of clauses stemming from SPASS. Nevertheless, the results are—as already mentioned—worse in comparison with the results obtained with contraction-based requests. A main reason for this is surely the vagueness of our criterion for estimating whether or not a clause is possibly needed in a proof. Thus, an interesting topic for further research would be to examine whether more complex criteria entails better results.

Contraction-based requests were often well-suited for exchanging clauses which are able to contract the search state. We underline this with two examples: When tackling the problem ROB011-1 the term $negate(add(x, y))$ occurred often as a sub-term of active clauses of DISCOUNT which generated many descendants. Then, SPASS was

able to response with the clause $negate(add(x, y)) = negate(add(y, x))$ which could be used for contracting 65 rules of DISCOUNT, i.e. nearly the whole set of rules could be simplified. Another example is `ROB016-1`. In the first cooperation phase SPASS detected that many clauses being involved in expanding inferences had sub-terms being instances of the term $negate(add(negate(add(x, negate(y))), negate(add(y, x))))$. The response $negate(add(negate(add(x, negate(y))), negate(add(y, x)))) = x$ of DISCOUNT could again be often used for rewriting. Hence, SPASS generated less unnecessary clauses and could find the proof faster.

All in all we can say that requirement-based cooperation indeed enables cooperative provers to outperform sequential provers. The orientation on demands of certain provers introduced by requests and responses does not only allow to decrease the runtimes for certain problems but is is also possible to solve problems none of the coupled provers can cope with.

7 Discussion and Future Work

State-of-the-art theorem provers have reached a considerable level of performance. Nevertheless, they suffer from the fact that usually no single strategy is able to deal with a large number of problems. This poses severe problems especially for an unexperienced user of a prover. Hence, cooperation of theorem proving strategies appear to be a promising approach in order to overcome this problem.

Our approach of realizing cooperative provers is requirement-based cooperative theorem proving. This method realizes some kind of demand-driven cooperation which is opposite to the commonly used success-driven approaches. Thus, it is possible to incorporate an orientation on the concrete needs of theorem provers into the cooperation scheme.

We described an abstract framework for requirements and particularly two certain aspects of requirement-based cooperation: On the one hand requirement-based exchange of facts, on the other hand sub-problem division and transfer via requests. Our experimental studies revealed that our concept indeed enabled the cooperative system to find proofs considerably faster than each prover when working alone.

There are some related approaches for transferring sub-problems as well as for exchanging facts among several provers.

A well-known approach for distributing sub-problems among various agents is the contract-net protocol [Smi80] from the area of multi-agent systems. Thus, we want to discuss the differences between the contract-net protocol and our approach of sub-problem transfer via requests.

The first difference is that we employ a very simple method to decide which prover should tackle a certain sub-problem: We distribute each sub-problem to an arbitrary prover in the network. In contrast, in the contract-net protocol sub-problems are sent to idle agents. Then, each agent computes the possible effort for solving it and sends this information to the distributor of the sub-problems. After that, each sub-problem is transferred to the agent that appears to be best-suited for solving it. As one can see,

our approach is less complex and requires also less communication. Since in theorem proving only very vague criteria could be used in order to estimate whether one prover is probably especially suited for solving a given problem our concept seems to be sufficient.

The second main difference is that in our approach there are never idle provers in the network that can be employed in order to tackle certain sub-problems. In contrast, the idea of our approach is that at any time each prover tackles mainly the original problem or a sub-problem that it has identified by itself. Sub-problems of other provers are tackled additionally by each prover, i.e. they have no higher priority.

Finally, we employ a time limit for the solution of sub-problems of other provers. This corresponds again to the idea of working mainly on own problems and only partly on sub-problems of other provers.

Related approaches for an exchange of facts between theorem provers are mainly—as already discussed—success oriented (e.g., [Sut92], [Den95], [BH95], [Bon96], [FD97]). In these approaches information is sent to other provers without considering specific needs of the receivers. A similar scheme of requests and responses for exchanging facts is—to our knowledge—only realized in the DARES system ([CMM90]). However, requests are in this context only needed to preserve completeness since no prover in the DARES system has all axioms in its set of start clauses: If a prover is not able to perform inferences and the empty clause has not yet been derived it imports, via requests, facts from other provers that allow it to continue its search. Hence, requests do not introduce a real kind of orientation on concrete needs of provers in the cooperating system.

Finally, interesting topics for future research are the following: Surely, our experimental studies should be further extended so as to obtain more reliable data. Furthermore, it would be interesting to integrate also analytic provers, e.g. tableau-style provers, into our cooperative system. Since these provers are based on a division of the original problem into sub-problems especially sub-problem transfer via requests might be promising. Then, analytic provers can be used for identifying and transferring sub-problems, saturation-based provers for solving or simplifying them. Thus, requirement-based theorem proving offers the possibility to integrate both top-down and bottom-up theorem proving approaches.

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A System For Distributed Equational Deduction. In *Proc. 6th RTA*, pages 397–402, Kaiserslautern, 1995. LNCS 914.

[BDP89] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion without Failure. In *Coll. on the Resolution of Equations in Algebraic Structures*. Academic Press, Austin, 1989.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BH95] M.P. Bonacina and J. Hsiang. The Clause-Diffusion methodology for distributed deduction. *Fundamenta Informaticae*, 24:177–207, 1995.

[Bon96] M.P. Bonacina. On the reconstruction of proofs in distributed theorem proving: a modified Clause-Diffusion method. *Journal of Symbolic Computation*, 21(4):507–522, 1996.

[BS97] F. Baader and K.U. Schulz(Eds.). *Applied Logic Series 3: Frontiers of Combining Systems*. Kluwer Academic Publishers, 1997.

[CMM90] S. E. Conry, D. J. MacIntosh, and R. A. Meyer. Dares: A distributed automated reasoning system. In *Proceedings of AAAI-90*, pages 78–85, 1990.

[Den95] J. Denzinger. Knowledge-based distributed search using teamwork. In *Proc. ICMAS-95*, pages 81–88, San Francisco, 1995. AAAI-Press.

[Der90] N. Dershowitz. A maximal-literal unit strategy for horn clauses. In *Proc. 2nd CTRS*, pages 14–25, Montreal, 1990. LNCS 516.

[DF94] J. Denzinger and M. Fuchs. Goal oriented equational theorem proving. In *Proc. 18th KI-94*, pages 343–354, Saarbrücken, 1994. LNAI 861.

[Ert92] W. Ertel. OR-Parallel Theorem Proving with Random Competition. In *Proceedings of LPAR'92*, pages 226–237, St. Petersburg, Russia, 1992. Springer LNAI 624.

[FD97] D. Fuchs and J. Denzinger. Cooperation in theorem proving by loosely coupled heuristics. Technical Report SR-97-03, University of Kaiserslautern, Kaiserslautern, 1997.

[Fit96] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 1996.

[Fuc97] M. Fuchs. Evolving combinators. In *Proc. CADE-14*, pages 416–430, Townsville, Australia, 1997. LNAI 1249.

[Smi80] R.G. Smith. The Contract-Net Protocol: High Level Communication and Control in a Distributed Problem Solver. *IEEE Trans. Comp., C-29*, pages 1104–1113, 1980.

[SS97] G. Sutcliffe and C.B. Suttner. The results of the cade-13 ATP system competition. *Journal of Automated Reasoning*, 18(2):271–286, 1997.

[SSY94] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In *CADE-12*, pages 252–266, Nancy, 1994. LNAI 814.

[Sut92] G. Sutcliffe. A heterogeneous parallel deduction system. In *Proc. FGCS'92 Workshop W3*, 1992.

[Wei93] C. Weidenbach. Extending the resolution method with sorts. In *Proc. IJCAI '93*, pages 60–65, Chambery, 1993.

[WGR96] C. Weidenbach, B. Gaede, and G. Rock. Spass & Flotter Version 0.42. In *Proc. CADE-13*, pages 141–145, New Brunswick, 1996. LNAI 1104.