Chapter 8

Conflict Resources

No one can serve two masters.
Either he will hate the one
and love the other,
or he will be devoted to the one
and despise the other.
J.C. in Matthew 6:24

In this chapter we extend the model used so far for DisCSP in order to accommodate access to a resource for several agents. Examples of addressed resources in DisCSPs are:

- variables (their labels/instantiations)
- ordering on agents
- computation resources

First we present a marking technique for ordered proposal sources. It allows for defining a total order, that induces a causal order within classes of messages that travel solely downward a hierarchy.

Two applications of this marking technique are described in the following two chapters: aggregations and reordering. A third application to sharing computing resources is detailed in (Silaghi & Faltings 2001). The known alternatives are also discussed.

8.1 Proposals

As mentioned in Section 4.2.1, one of the drawbacks of ABT is that it does not support cooperation among more than two agents that share a resource. The real problem comes from the fact that the different agents should be able to coherently agree on a common proposal. I assume reliable channels and correct processes.

A decision can be taken by giving different priority to distinct agents. However, this would simply lead to the loss of any proposal coming from lower priority agents and the problem of ABT is not solved. I propose to generalize the notion of proposal in a way that allows several proposals to be composed.

Definition 8.1 *A proposal of an agent on a shared resource is composed of a sample requirement and a set of alternative requirements. The sample requirement is the default when nobody changes it. The set of alternative requirements is the set of proposals allowed for lower priority agents that share the same resource.*

A relation, *complies to*, can be defined on proposals. It specifies whether a given proposal made by a lower priority agent complies with a proposal made by higher priority agents. It should

be noted that a given proposal of a lower priority agent may comply with a proposal of a higher priority agent, unintentionally. In order to simply treat proposals in a coherent manner, they can be marked in a way that allows to check easily their intended "complies to" relationship.

Definition 8.2 *We say that the relation "p1 complies to p2" between two proposals is* intended *if the agent generating p1 explicitly states this relation.*

8.2 Signatures

Now I introduce a *marking technique* that allows for the definition of a total order among the proposals made concurrently and asynchronously by a set of ordered agents on a shared resource (e.g. an order, a variable, processor time). This technique also allows to specify implicit "complies to" relations.

Definition 8.3 *An* **abstract agent** *is an entity (e.g. physical agent, community of agents, behavior or routine), that from a logical point of view can be globally modeled as a problem solving agent (e.g. a state machine). It can be seen as a role or office accomplished by an agent, a set of agents, etc.*

In this chapter, $\mathcal{R}$, denotes a generic shared resource.

Definition 8.4 *A* **proposal source** *for a resource $\mathcal{R}$ is an entity (e.g. an abstract agent) that can make specific proposals concerning the allocation (or valuation) of $\mathcal{R}$.*

We will consider that a total order $\prec$ is defined on *proposal sources*. The *proposal sources* with lower position according to $\prec$ have a higher priority. The *proposal source* with position k is noted $P_k^{\mathcal{R}}$, $k \geq k_0^{\mathcal{R}}$. $k_0^{\mathcal{R}}$ is the first position.

Definition 8.5 *A* **conflict resource** *is a resource for which several agents can make proposals in a concurrent and asynchronous manner.*

Each *proposal source* $P_i^{\mathcal{R}}$ maintains a counter $\mathbf{C_i^{\mathcal{R}}}$ for *conflict resource* $\mathcal{R}$. Markers for defining order on messages in distributed settings have been first introduced in (Johnson & Thomas 1975). The markers involved in our *marking technique for ordered proposal sources* are called **signatures**.

Definition 8.6 *A* **signature** *is a chain h of pairs, $|a{:}b|$, that can be associated to a proposal, Z, for $\mathcal{R}$. A pair $p{=}|a{:}b|$ in h signals that Z complies to a proposal for $\mathcal{R}$ that was made by $P_a^{\mathcal{R}}$ when its $C_a^{\mathcal{R}}$ had the value b, and it knew the prefix of p in h.*

Example 8.9 *The next are examples of signatures:*

- $||$ *-an empty signature*
- $|1{:}5|$ *-a one pair signature*
- $|1{:}5|2:4|$ *-a two pairs signature*
- $|2{:}4|3:2|5:2|7:23|$ *-the pairs are ordered in a signature.*

While the form of our pairs resembles the ones in (Johnson & Thomas 1975), their semantic is different. An order $\propto$ (read "follows") is defined on pairs such that $|i_1{:}l_1|\propto|i_2{:}l_2|$ if either $i_1{>}i_2$, or $(i_1{=}i_2)\wedge(l_1{<}l_2)$ — this is due to the fact that l_1 and l_2 are counters.

Example 8.10 *The next are examples of order on pairs:*

- $|1{:}5| \propto |1{:}7|$
- $|2{:}5| \propto |1{:}3|$
- $|4{:}5| \propto |2{:}5|$

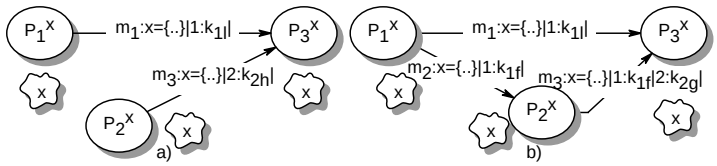

Figure 8.1: Simple scenarios with messages for proposals on a resource, x.

- $|4{:}7| \propto |2{:}5|$

Definition 8.7 *A signature h_1* **is stronger than** *a signature h_2 if a lexicographic comparison on them, using the order $\propto$ on pairs, decides that $h_2 \propto h_1$.*

Example 8.11 *The next are examples of order on signatures:*

$\|1{:}7\|$	*stronger than*	$\|2{:}8\|$
$\|2{:}9\|$	*stronger than*	$\|2{:}8\|$
$\|1{:}7\|3:5\|$	*stronger than*	$\|1{:}7\|$
$\|1{:}7\|3:5\|$	*stronger than*	$\|1{:}7\|4:7\|$
$\|1{:}7\|3:6\|$	*stronger than*	$\|1{:}7\|3:5\|$
$\|1{:}7\|3:5\|4:2\|$	*stronger than*	$\|1{:}7\|3:5\|$
$\|1{:}7\|3:6\|$	*stronger than*	$\|1{:}7\|3:5\|4:3\|$
$\|1{:}7\|3:6\|4:2\|$	*stronger than*	$\|1{:}7\|3:5\|4:3\|$

This is a generalization of the notion *stronger* on assignments. $P_k^{\mathcal{R}}$ builds a signature for a new proposal on $\mathcal{R}$ by prefixing to the pair $|k{:}l_{k^j}|$, the strongest signature that it knows for a proposal on $\mathcal{R}$ made by any $P_a^{\mathcal{R}}$, $a{<}k$. l_{k^j} is the current value of $C_k^{\mathcal{R}}$. The $C_a^{\mathcal{R}}$ in $P_a^{\mathcal{R}}$ is reset each time an incoming message announces a proposal with a stronger signature, made by higher priority *proposal sources* on $\mathcal{R}$. $C_a^{\mathcal{R}}$ is incremented each time $P_a^{\mathcal{R}}$ makes a proposal for $\mathcal{R}$.

Definition 8.8 *A signature h_1 built by $P_i^{\mathcal{R}}$ for a proposal is* **valid** *for an agent A if no other signature h_2 (eventually known only as prefix of a signature h_2') is known by A such that h_2 is stronger than h_1 and was generated by $P_j^{\mathcal{R}}$, $j \leq i$.*

For example, in Figure 8.1 the agent P_3^x may get messages concerning the same resource x from P_1^x and P_2^x. In Figure 8.1a, if the agent P_3^x has already received m_1, it will always discard m_3 since the *proposal source* index has priority. However, in the case of Figure 8.1b P_2^x knows $|1{:}k_{1f}|$ and the message m_1 is the strongest only if $k_{1f}{<}k_{1l}$. The length of a signature tagging proposals for a *conflict resource*, $\mathcal{R}$, is upper bounded by the number of *proposal sources* for $\mathcal{R}$.

I also define some notations:

- The strongest signature received by $P_k^{\mathcal{R}}$ up to and including a given event is denoted by signature(k), and
- the signature marking a message m is denoted signature(m).

8.2.1 Alternatives to signatures

Alternatively to using signatures, proposals could be tagged using a simple counter. In this case an agent needs to store the last proposals on $\mathcal{R}$ made by each predecessor proposal source and considers as current proposal a combination of them. Then $P_a^{\mathcal{R}}$ needs not resend its old proposal p when p remains consistent with the view of $P_a^{\mathcal{R}}$ that changes. Instead $P_a^{\mathcal{R}}$ would have to send a new proposal if its proposal changes to become identical with the strongest received proposal.[1] This is a tradeoff and the best alternative depends on the problem at hand.

In Section 10.4 I give an example where this technique is natural.

[1] This problem can in its turn be solved by considering that such actions are implicit in case of conflict.

8.2.2 Causal order for downward messages

We start recalling a few definitions introduced in (Lamport 1978).

Definition 8.9 *Given a set of totally ordered messages, we call* downward message *a message m whose sender has a higher priority than its destination.*

Definition 8.10 *An event e_1 happens before another event e_2 if either:*

- *e_1 is the event of sending a message m, whose reception is the event e_2.*
- *e_1 and e_2 happen on the same process p, and e_1 happens earlier than e_2 in p.*
- *It results using the transitive closure of the previous two criteria.*

Consider a system composed of a set of totally ordered agents (processes) and a global history defined by a set of downward messages tagged with signatures. The only events taken into account in the next theorem are the reception and emissions of messages.

Theorem 8.1 *If there exists a casual chain of events on which send(m_1) happens before send(m_2), then the signature of m_1 is stronger than the one of m_2.*

Proof. Given a process p (p-th according to $\prec$), signature(p) is modified on two types of events:

- when a message m is sent after sending m', $C_p^{\mathcal{R}}$ is incremented and:
  - either signature(p) has been modified by a received message and a new pair, |p:0|, is appended to signature(p) in the new message, or
  - the last pair of signature(m), |p:$C_p^{\mathcal{R}}$|, follows the one of m'.

  In both cases, signature(m) is stronger than signature(m').
- A new downward message, m, is delivered to p on the casual chain and:
  - either signature(p) is identic or stronger than signature(m) and m is discarded, or
  - signature(m) is stronger than signature(p). In this case, m being a downward message, signature(p) is replaced by signature(m).

In both cases, the signature of the message sent next by p, signature(p)|p:$C_p^{\mathcal{R}}$|, becomes stronger than signature(m).

Given two distinct events e_1 and e_2 found in a prefix of a local history of a process p. Since signature(p) evolves only into a stronger signature, signature(e_2) is stronger than signature(e_1).

Given a message m sent by a process p_1 to a process p_2, signature(receive(m)) is assigned the strongest signature between signature(m) and signature(p_2). Therefore signature(receive(m)) is stronger than signature(m) which is in its turn stronger than signature(p_1).

Due to the transitivity of the lexicographical comparison, the stronger relation is transitive. Therefore, the messages whose emission events are on a causal chain defined by "happens before", are tagged with signatures that are ascending according to the stronger relation. □

The following concept is well-known:

Definition 8.11 *Causal order delivery is a mechanism for delivering messages to agents according to a causal order on them.*

As noticed in (Luo *et al.* 1992), casual order delivery is not necessary in asynchronous search, and it is sufficient to discard delayed messages.

8.2.3 Extensions to signatures

Signatures have been defined for a set of totally ordered proposal sources. However, their use can be extended in a straightforward manner to a tree path of downward messages. Such a tree could be found in DFS hierarchies of CSP problems (Collin *et al.* 1991a). The PAS algorithms that are described in (Silaghi & Faltings 2001), provide a working example of such an extension.

8.3 Parallel runs on computing resources

In (Silaghi & Faltings 2001) we have introduced another application for signatures, an application that we did not yet fully explore. It has been early realised that agents are often idle in asynchronous search (Luo *et al.* 1992). One well-known source of inefficiency is the lack of load balance. (Luo *et al.* 1992) has already noted that in ABT, higher priority agents have very little work to do in comparison with the lower priority ones. The solution proposed by (Luo *et al.* 1992) and reused in (Solotorevsky *et al.* 1996a) consists in allowing lower priority agents to precompute new local solutions in their idle time. While for simple local problems, this does not apply, for hard local problems it can be useful as long as the local problem is not exausted. Because of space complexity, only a bounded number of local solutions can be precomputed.

(Hamadi 1999b) came with a new solution for a better load balance. Hamadi has proposed to allow the agents to lead in parallel several independent distributed computations on domain-based disjoint parts of the problem (Luo *et al.* 1992). The different computations can cooperate when the agents use common storage for nogoods. We have arrived to a similar technique in (Silaghi & Faltings 2001) and we have been told only later about the earlier work in (Hamadi 1999b).

The basic idea in (Hamadi 1999b; Silaghi & Faltings 2001) is to define a bounded number of *slots* for parallel distributed computations. A distinct independent distributed search process runs in each slot. The number of slots has to be bounded in order to make sure that the space requirement in each agent can remain polynomial. Moreover, for the same reasons for which a number of threads that is higher than the number of processors is harmful, a number of slots higher than the ratio of idle agent time leads to unacceptable management overhead.

In (Hamadi 1999b), the global problem is statically distributed to slots. One of the main concerns in (Silaghi & Faltings 2001) is to allow redistribution of the problem. Dynamic redistribution of problems has been often used for domain-based distributed computations. It is highly probable that some branches of the search are easier than others, and some slots can prove quickly to be infeasible.

We define a hierarchy of the slots and one abstract agent receives the task of managing the top level in this hierarchy. Whenever a slot proves infeasibility for its problem, the corresponding nogood is sent to the first branch of the slot hierarchy where this slot distinguish itself from other slots. When a slot is available, its availability is broadcasted and different agents can make proposals defining the allocation of the available slot. All messages are tagged with slot signatures, therefore allocation will be coherently adopted. The signatures of the slots are expected to be short for problems with large domains when only few high priority agents have chances to win with their allocation proposals. This is an example for the generalization of signatures from totally ordered proposal sources to a tree hierarchy of proposal sources.

8.4 Consensus for proposing instantiation of variable

As mentioned above, complete asynchronous search algorithms for DisCSPs require an order on participating agents. This order has been often perceived as strongly harming fairness. However, already in the original presentation of asynchronous backtracking (ABT) agents stand for variables. We show here that an additional level of abstraction in ABT, namely replacing ABT agents with a *cooperation Mechanism*, can lead to important shifts in fairness properties. It is remarkable that the result can be achieved without losing completeness and with minimal adaptations to the distributed protocol. The result of this section comes out of joint research with Ion Constantinescu.

8.4.1 Introducing coordination

Our objective is to explore ways for allowing the process of instantiation a variable to be controlled by two or more agents. For that we propose an extension to the model described above with the concepts of *coAgent* and *coVariable* (cooperation agent/variable). First, the assignment process itself is going to be exterior to the ABT algorithm and we are going to refer to it as the *coMechanism*. A *coMechanism* for deciding instantiations with interesting behavior could consist

of a voting mechanism over the set of values obtained with unanimity when the acceptable sets of all agents are intersected.

First we present a basic set of relations between existing and proposed concepts and then we discuss a number of choices for completely defining this model.

Any *coAgent* controls the instantiation of exactly one *coVariable*. A DisCSP can have a number of zero or more *coAgent*s / *coVariable*s.

The case of a DisCSP with zero *coAgent*s is exactly the initial case presented above in Section 4.2. The case of a DisCSP with two or more *coAgent*s can be easily derived from the case of a DisCSP with one *coAgent*. As such we are going to consider next the case of a DisCSP with one *coAgent* / *coVariable*.

The agents that participate to the *coMechanism* are said to *support* the *coAgent* or otherwise are called *supporting Agent*s. The agents that participate in the **ABT** are called *ABT Agent*s.

Further defining the newly proposed concepts raises a number of questions:

- 1. what is the intersection set between the *supporting Agent*s and *ABT Agent*s?
- 2. how are the agents controlling the *coMechanism* interacting with the agents participating to the **ABT**?
- 3. is an agent controlling the *coMechanism* and thus participating in the instantiation of the *coVariable* allowed to control also a local variable?

Our first proposed model gives the following responses:

- 1. only one agent acts as both a *supporting Agent* and a *ABT Agent* (the cardinality of the intersection set from 1. is one). This is actually going to be the *coAgent*.
- 2. the *coAgent* is going to act as a gateway running the **ABT** and also the *coMechanism*
- 3. the *coAgent* doesn't control a local variable

Figure 8.2 shows an example of running the above algorithm with *coAgent* A_2.

In step (a) agent A_1 instantiate his variable to 1 and sends a **ok?** message to the *coAgent* A_2 . Upon receiving the **ok?** message agent A_2 chooses a possible value for his local variable - in our case 2 - and initiates the *coMechanism* with this variable. The *coMechanism* returns successfully and as such agent A_2 assigns the results to his current_value ($current\_value = 2$).

In step (b) A_2 sends an **ok?** message to A_3 with the newly assigned value.

In step (c) A_3 is not able to instantiate his local variable due to incompatibility with the current value of the higher priority agent A_2 so it *backtrack* by sending back to A_2 a **nogood** message.

In step (d) A_2 has his local variable incompatible with his nogood-list and is also not able to find a new assignment. As such it is also forced to backtrack and sends back to A_1 a **nogood** message.

8.4.2 Abstract agents

As noticed in the example, the *coAgent*s in the previous algorithm can become bottlenecks in the interaction between ABT and *coMechanism*. We propose now another formalism, ShABT, that modifies ABT but removes the bottlenecks and brings additional fairness properties. A total order is defined on the public variables of the DisCSP. ABT is modified as follows.

Each physical agent A_i participates under the role/name A_{k_i} in the instantiation of each of its variables x_k and communicates the variable value to other interested agents. A_{k_i} is referred to as a *coAgent* and enforces the constraints of A_i involving x_k and higher priority variables. The ABT

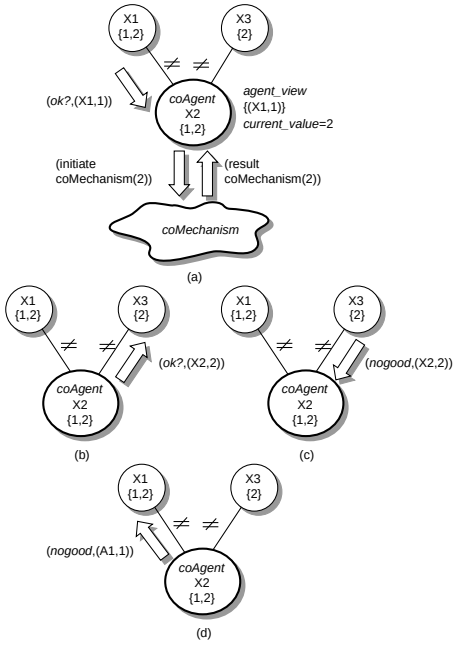

Figure 8.2: ABT algorithm with *coAgent* and *coMechanism*

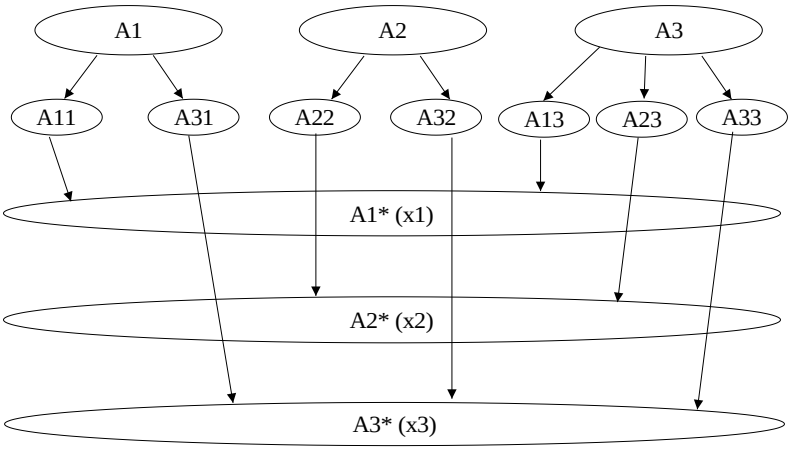

Figure 8.3: ShABT agent hierarchy. A1, A2, A3 are physical agents. A11, A31 ,... are *coAgent*s. A1*, A2*, A3* are the abstract agents for x_1, x_2, respectively x_3.

agents are replaced by abstract agents. The abstract agent for x_i is denoted A_{i_*}. The abstraction hierarchy for the previous problem is shown in Figure 8.3.

Each *coAgent* holds a list of *outgoing links* represented by a set of physical agent names. ShABT assumes that every link is directed from the value sending agent to the constraint-evaluating-agent. Each physical agent owns an *agent view* that can be accessed by all its *coAgent*s.

Based on their constraints, the *coAgent*s perform inferences concerning assignments in their *agent view* and explanations of disagreement of other *coAgent*s, received via *coMechanism*. To send a message to the abstract agent A_{i_*}, it is sent to a *coAgent* A_{i_t}, for some t.

The following types of messages are exchanged in ShABT:

- **ok?** message transporting an assignment is sent to a nogood owner physical agent to ask

```
when received (ok?,⟨x_j, d_j⟩) do
    add(x_j,d_j) to agent view;
    check_agent_view;
when received (nogood,A_{j_t},¬N) do
    Integrate-nogood(¬N);
    old_value ← current_value;
    check_agent_view;
    when old_value = current_value
22.1    send (ok?,⟨x_i, current_value⟩) to A_t;
procedure Integrate-nogood(¬N) do
    when ⟨x_k, d_k⟩, where x_k is not connected, is contained in ¬N
        send add-link to A_{k*};
        add ⟨x_k, d_k⟩ to agent view;
    put ¬N in nogood-list;
22.2 add other new assignments to agent view//pShABT;
procedure check_agent_view do
    when agent view and current_value are not consistent
22.3    select d=coMechanism(++lcm,agent view,D_i,nogood-list,data);
        if d = ∅ then
            wait and get nogoodlist(nl);
            foreach (n ∈ nl) do
                Integrate-nogood(n);
            backtrack;
        else
            current_value ← d;
            send (ok?,⟨x_i, d⟩) to lower priority agents in outgoing links ;
procedure backtrack do
    nogoods ← {V | V = inconsistent subset of agent view};
    when an empty set is an element of nogoods
        broadcast to other agents that there is no solution;
        terminate this algorithm;
    for every V ∈ nogoods do
        select ⟨x_j, d_j⟩ where x_j has the lowest priority in V;
22.4    send (nogood,A_{i_z},V) to A_{j*};
        remove (x_j,d_j) from agent view;
    check_agent_view;
```

Algorithm 22: Procedures of A_{i_z} for receiving messages in ShABT.

whether a chosen value is acceptable. The reception event is triggered for all *coAgent*s.

- **nogood** message transporting a *nogood*. It is sent from the *coAgent* that infers a *nogood* $\neg N$, to a constraint-evaluating-*coAgent* for $\neg N$.
- **add-link** message announcing A_{i_k} that the sender A_j owns constraints involving x_i. A_{i_k} inserts A_j in its *outgoing links* and answers with an **ok?**.
- **coMechanism** message announcing A_{i_t} that the sender A_{i_j} runs the *coMechanism* for the assignment of x_i. It transports the current round ID of the *coMechanism* on the variable, as well as the data needed for the *coMechanism*.
- **nogoodlist** messages are exchanged within the *coMechanism* for composing *nogood-lists* after

```
procedure coMechanism(cm,agent view,D_i,nogood-list,data) do
    lcm ← max(cm,lcm);
    discard coMechanism data for rounds less than lcm;
    launch d=coMechanism by sending (coMechanism,A_{i_h},cm,local-data) to all agents A_{i_t};
    block waiting for coMechanism to end;
    compute and return d;
when received (coMechanism,A_{i_h},cm,data) do
    d = coMechanism(cm,agent view,D_i,nogood-list,data);
    if (d ≠ ∅) then
        current_value ← d;
        send (ok?,⟨x_i, d⟩) to lower priority agents in outgoing links;
    else
23.1    send (nogoodlist,A_{i_h},nogood-list);
```

Algorithm 23: Procedures of A_{i_z} for *coMechanism*.

failure.

The agents start by launching *coMechanism* (Algorithm 23) for instantiating their variables concurrently. The agents answer to received messages according to the Algorithm 22. *coMechanism* rounds are tagged with monotonically increasing IDs. **coMechanism** messages carry the round ID and the local data needed for *coMechanism*. E.g., for the *coMechanism* mentioned in the previous section, the data can consists of the set of available values and a preference (e.g. a number from 0 to 1) for each of them. The winning assignment will belong to the intersection of the available values received for the current round from all *coAgent*s. It can be chosen by optimizing a function (e.g. sum) over the preferences of all agents. For ensuring consensus, ties can be broken with an order on physical agents.

To completely avoid an order on physical agents, one can restrict the preferences of the agents to distinct disjoint sets of prime numbers and use the multiplication as function to optimize. Similar effects can be obtained with the sum as function, and with predefined discrete sets of numbers as preferences.

One can envisage the use of cryptographic voting mechanisms for enhancing privacy for preferences during the variable assignments. Such protocols are the Merritt's election (Merritt 1983) and Benaloh's fault-tolerant election (Benaloh 1987; Cramer *et al.* 1996).

Theorem 8.2 *ShABT is correct, complete and terminates.*

Proof. The proof is identical with the proof of ABT in (Yokoo *et al.* 1992), where the reasoning is performed on variables. □

8.4.2.1 Polynomial space ShABT

ShABT has exponential space requirements, but a polynomial space version called pShABT can be obtained as follows:

- Instantiations have to be tagged (e.g. with the ID of *coMechanism*). Only the most recent is retained and old ones are invalidated.
- The line 22.2 can and has to be called.
- Only one valid nogood has to be stored for a value.

Theorem 8.3 *pShABT is correct, complete, terminates and has polynomial space requirements.*

Proof. The proof is identical with the proofs for polynomial space versions of ABT, where the reasoning is performed on variables. □

8.4.2.2 Optimizations

A lot of optimizations are possible in both ShABT and pShABT. For example, the *coMechanism* call at line 22.3 can be avoided if a nogood can be locally inferred. Especially for ShABT, but also for pShABT, at line 23.1, *coAgent*s can send a resume nogood (e.g. as the conflict list CL in AAS). Otherwise, for ShABT it becomes important to mark already sent nogoods and to avoid sending them again to the same target.

8.5 Summary

In this chapter we have introduced a marking technique called signatures, allowing several totally ordered proposal sources (e.g. agents) to simultaneously and concurrently make proposals for cooperating to the coherent allocation of a shared resource. While other techniques could be used for achieving similar flexibility, the signatures are a very simple and promising solution.

Two important applications of the signatures, namely aggregation and reordering, are described in the following two chapters.

We end describing another possible application, namely the allocation of computing resources. This application is important for illustrating the generality of the new marking technique.

An alternative for allowing agents to share control of a variable is to decide values with a consensus mechanism (see Section 8.4). While this brings some fairness among decisions for values of a variable, this remains unfair in general, as agents interested in other lower priority variables are disfavored. From a technical point of view, in order to avoid bottlenecks, one need to avoid communicating consensus results via a single agent. The solution that we propose in (Silaghi & Constantinescu 2002) is to reformulate ABT by retaining its logical mechanism but logically replacing agents with sets of agents. Each agent A_i is logically replaced in the ShABT protocol with the set of agents that know constraints on x_i, denoted A_{x_i}. Any message sent in ABT to an agent A_i can be sent to any agent in the set of agents deciding x_i. The ShABT mechanism used by agents in A_{x_i} for allowing a behavior equivalent to the one of a single agent, A_i, in ABT is called coMechanism. Details of the implementation that we propose are described in (Silaghi & Constantinescu 2002).