REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
08-10-2003	Final Report	16 September 2002 - 09-Sep-04

4. TITLE AND SUBTITLE

Monitoring and Meta-Reasoning in Multi-Agent Systems

5a. CONTRACT NUMBER
FA8655-02-M4056

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Professor Vladimir Marik

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Czech Technical University of Prague
CTU FEL
Technická 2,
Praha 6 166 27
Czech Republic

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0014

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
SPC 02-4056

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report results from a contract tasking Czech Technical University of Prague as follows: There will be four aspects to this research. i) the role of meta reasoning in multi-agent systems will be analyzed; ii)approaches for knowledge acquisition will be investigated; iii) methods of knowledge analysis and hypothesis formation about other agents will be determined iv) results will be implemented and tested.

15. SUBJECT TERMS
EOARD, Multi Agent Systems, Agent Based Systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18, NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			JAMES GREENBERG
UNCLAS	UNCLAS	UNCLAS	UL		19b. TELEPHONE NUMBER *(Include area code)* +44 (0)20 7514 4922

Monitoring and Meta-Reasoning in Multi-Agent Systems

MRinMAS

contract number:
FA8655-02-M4056

principal investigators:
Michal Pěchouček and Vladimír Mařík

team members:
Jaroslav Bárta, Jan Tožička,
Olga Štěpánková and Michal Jákob

Gerstner Laboratory,
Czech Technical University in Prague

http://gerstner.felk.cvut.cz

Contents

Summary. Agent's meta-reasoning is a computational process that implements agent's capability to reason on a higher level about another agent or a community of agents. There is a potential for meta-reasoning in multi-agent systems. Meta-reasoning can be used for reconstructing agents' private knowledge, their mental states and for prediction of their future courses of action. Meta-agents should have the capability to reason about incomplete or imprecise information. Unlike the ordinary agents, the meta-agent may contemplate about the community of agents as a whole. This contribution presents application of the meta-reasoning process for the agent's private knowledge detection within the multi-agent system for planning of humanitarian relief operations. Three pivotal meta-reasoning technologies are discussed in the report: **simulation**, **deduction** and **induction**.

1

Introduction

Multi-agent systems are collections of autonomous, heterogeneous agents with specialized functionalities. The agents are usually able to carry out collective decision making, share resources, integrate services or just collaboratively seek for specific information. The possible application domains of such multi-agent systems are e.g. production planning and scheduling, supply chain management, simulation of virtual enterprisers, or coalition formation processes. Distributed problem solving architectures provide important features, e.g. capability to find 'reasonably-good' solutions efficiently, robustness and a very high degree of fault-tolerance, reconfigurability capabilities, 'openness' of the community to integrate new agents or to replace the disappearing, etc.

We have demonstrated [1] that the concept of the multi-agent system is appropriate for planning humanitarian relief operations. The agents representing the humanitarian organizations control their actions with respect to their private restrictions. In this contribution, we focus on detection of these constrains by monitoring the community communication. We use a formal model of meta-agent presented in [2] for implementation of the meta-reasoning using different methods of artificial intelligence.

This chapter introduces the target domain for meta-reasoning activities and basic meta-reasoning terms. In chapter 2, we describe the abstract meta-reasoning architectures and formal description of the reasoning processes. In chapters 3 and 4 we describe the methods used in meta-reasoning implementation.

1.1 Meta-Reasoning and Meta-Agents

We refer to meta-reasoning as an agent's capability to reason about the knowledge, mental states and reasoning processes of other members of the multi-agent community. We will refer to **object agents** which are subject of another agent's meta-reasoning process. Meta-reasoning can be carried out either by the object agent or by a specific agent, whose role is only to carry out the

meta-reasoning related process. We will refer to **meta-agent** as to any agent with the meta-reasoning ability.

According to the role/contribution of the object agents in the community to the meta-reasoning process we distinguish between two different types of meta-reasoning:

- **collaborative meta-reasoning**: In this case, the object agents are aware of being monitored, which is what they agree with and support. The purpose of collaborative meta-reasoning is very often an improvement of the object-agents' collective behavior.
- **non-collaborative meta-reasoning**: In this case, the object-agents do not want to be monitored and are not supporting the meta-reasoning process. The meta-reasoning agent is supposed to use different techniques rather than rely on the object agents to provide copies of their communication.

  Discussion of other approaches [3, 4, 5, 6] can be found in [7].

1.2 CPlanT Multi-agent System

Planning humanitarian relief operations [1] within a high number of intensively collaborating and vaguely linked non-governmental organizations is a challenging problem. Owing to the very special nature of this specific domain, where the agents may eventually agree to collaborate but are very often reluctant to share their knowledge and resources, we have combined classical negotiation mechanisms, teamwork theory [8] with the acquaintance models and social knowledge techniques [9].

CPlanT multi-agent system for planning humanitarian relief operations consists of three specific classes of agents: *resource-agents* represent the in-place resources that are inevitable for delivering humanitarian aid (e.g. roads, airports), *in-need agents* represent the centers of conflict that call for help (e.g. cities) and *humanitarian agents* represent the participating humanitarian agencies, which contribute to humanitarian aid missions. In this report we will be referring exclusively to the *humanitarian agents* when talking about object agents.

1.3 Cooperation Structures

We will work with two key cooperation structures that will be explained below: alliance and coalitions.

Agents can create a **coalition** – $\chi(m)$, a set of agents, which agreed to cooperate on a single, well-specified mission – m. The coalition is temporary and dissolves upon completion of the mission. When forming a coalition agents

need to propose collaboration, bid, and negotiate. During these activities substantial part of agents private knowledge discloses.

In order to optimize the amount of the disclosed private knowledge and to create an acceptably good coalition in a reasonable time, the community of object agents is partitioned into several disjoint groups called **alliances** – λ. All agents in an alliance agree to cooperate together, even if they can refuse to participate in a coalition allocated for a given mission (see section A.1).

In an ideal case, the alliance members solve the requested mission exclusively within the alliance. In cases when this is not possible, they need to subcontract part of the mission to alliance nonmembers (members of different alliances). All participating agents are still members of the coalition. Within the report we will refer the concept of **team** when we partition the coalition according to alliance membership. The coalition members from one alliance are regarded as members of the same team.

Agents' knowledge has been classified as (i) **private** – $\mathsf{K}_{\mathsf{pr}}(A)$, the piece of knowledge that is not accessible to any other agent (ii) **public** – $\mathsf{K}_{\mathsf{p}}(A)$, knowledge that is widely accessible to all agents and (iii) **semi-private** – $\mathsf{K}_{\mathsf{s}}(A)$, knowledge accessible reciprocally to alliance members only. For formal definition of these classes of knowledge see [1]. This concept is very closely related to the concept of agent's neighborhood [9].

1.4 Acquaintance Models

According to the model of social knowledge [9], the **agent's social neighborhood** $\mu(A)$ is a collection of agents that are subject of agent's A meta-reasoning processes. While $\mu^+(A)$ is a set of agents, which are monitored by the agent A, $\mu^-(A)$ is a set of all agents that monitor the agent A. Provided that A_1 denotes any agent (including meta-agents), we can say that:

$$\forall A \in \mu^-(A_1) : A_1 \in \mu^+(A). \tag{1.1}$$

However, in the multi-agent environment it is hard to implement a mutually shared knowledge structure that would represent agent's social neighborhood. This is why each agent maintains its social neighborhood by itself in a special knowledge structure – **acquaintance model** (see [9]). Therefore, we shall interpret the set $\mu^+(A)$ as a collection of agents that the agent A intentionally monitors, and the set $\mu^-(A)$ as a collection of agent's about which the agent A knows that they monitor it.

Under such interpretation, the formula (1.1) is true in both the collaborative and the non-collaborative environments, while the inverse formula $\forall A_1 \in \mu^+(A) : A \in \mu^-(A_1)$ is true in a collaborative environment only (as there cannot be agents which are monitored without knowing about it).

One of the meta-reasoning tasks is to reconstruct agents' acquaintance models where their semiprivate knowledge is stored. In CPlanT implemen-

tation, information about the resources, that the agent contributes with, is encoded as semi-private knowledge.

1.5 Decision Making in CPlanT

1.5.1 Forming a Coalition

In the CPlanT coalition planning system, the agents try to collaboratively form coalition that will work together on a specific mission. The **mission** specifies properties of a requested humanitarian operation in terms of a type of an operation (e.g. natural disaster, conflict, ...), severity, location, and primarily a list of requested services (food provision, shelters provision, ...).

$$m = \langle \texttt{id},\ \texttt{type},\ \texttt{degree},\ \texttt{location},\ \{\tau_i\}\rangle \tag{1.2}$$

For the mission – m to be accomplished the corresponding services – $\{\tau\}$ need to be implemented. The coalition is thus a collection of agents who commit themselves to participation in the mission by providing the requested services.

Once there is a request for mission operation, not all agents in the community are asked to form coalitions. Relevant agents subscribe the scenario map for notification about disasters and respective missions. Those agents, given the estimates of available resources of peer alliance members, construct **coalition proposal** – $\chi^*(m)$. The coalition proposal consist of the list of possible coalition members, the overall objective function of the coalition (e.g. price, delivery date, ...) and the list of required services that cannot be provided from within the alliance.

$$\chi^*(m) = \{A_i\}, \tag{1.3}$$

where an agent A_i is expected to provide a service τ_i of the mission m. Here we assume that a service is non-decomposable and can be implemented exclusively by a singular agent. Let us consider functions $\mathsf{leader}(\chi^*(m))$, $\mathsf{price}(\chi^*(m))$, $\mathsf{due}(\chi^*(m))$ and $\mathsf{to\text{-}do}(\chi^*(m))$ for giving properties of the proposal. These properties give an objective function that optimizes the coalition proposal selection process.

After the coalition proposal is specified the members of $\chi^*(m)$ enter a rather complicated negotiation process (within and outside the alliance that the coalition leader is a member of) in order to fix a joint commitment that will ultimately form the coalition $\chi(m)$. The coalition that will cover a specific mission is constricted in three steps:

1. **Coalition Leader Selection.** Subscribed agents then inform each other about objective function of their proposals (a function of $\mathsf{price}(\chi^*(m))$, $\mathsf{due}(\chi^*(m))$ and $(|\mathsf{to\text{-}do}(\chi^*(m))|)$ and compete one another. Under an assumption that the agents are true telling, they allocate (using a simple

bidding strategy) a **coalition leader** – an agent who covers the most within its own alliance (the most preferred criterion) and with the shortest delivery time (see figure 1.1). In our experiment we have allowed only a limited number of agents to be subscribed for a single location in the map. As the bidding strategy is based on broadcasting we cannot easily scale this solution for higher number of agents.

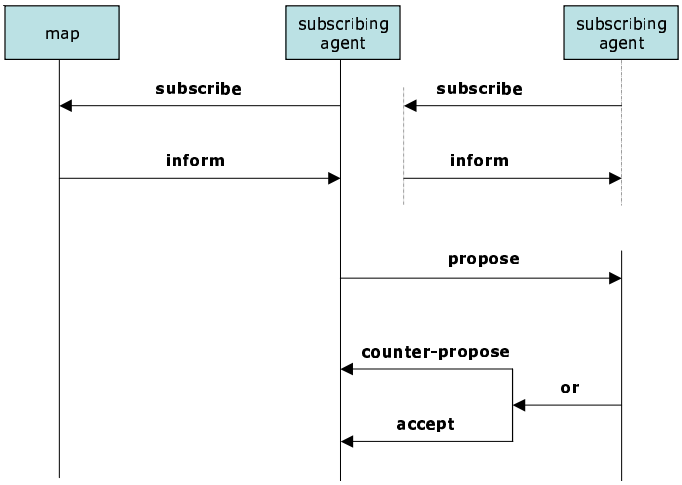

Fig. 1.1. Schema of interagent communication.

2. **In-alliance Coalition Formation.** The coalition leader tries to form a part of the coalition from his alliance members. Given the knowledge in its acquaintance model, the coalition leader directly requests the agents for (i) participating in a mission that will take place in specific place and will be coordinated by the specific coalition leader and (ii) providing the required resources (see figure 1.2). While the coalition leader knows about resources availability, it is not aware of agents' private knowledge that may restrict it to work under certain agents' leadership (eg. army unit) or a specific place (e.g. place with major population of muslims)
3. **Inter-alliance Coalition Formation.** Coalition leader tries to subcontract – using the contract net protocol – other agents to contribute with services the remaining requested services. In order to lower down the communication burden we will not want the coalition leader to do complete broadcasting. Instead only one member of each alliance (for simplicity we will refer in this report to this agent as a team-leader) is asked for the resources on behalf of the entire alliance. (see figure 1.3). Team leader provides the coalition leader with a suggested service provider(s). Upon an approval from the coalition leader, the team leader is asked to request

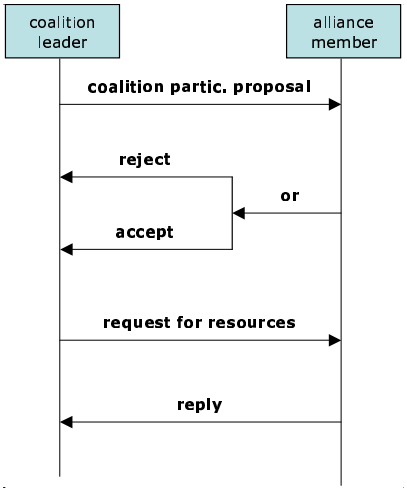

Fig. 1.2. Schema of interagent communication.

the resources from the suggested provider(s) in the same manner as above.

1.5.2 Goals of Meta-Reasoning

When re-constructing agents decision-making models we are interested in (i) how a coalition proposal can be created and (ii) how the actual coalition can be negotiated. Therefore, we have to monitor and reason about a coalition formation process (forming $\chi^*(m)$, answering a question of a type '*what-coalition-will-be-proposed?*') and find out whether agents will eventually provide requested services (forming $\chi(m)$, answering a question of a type '*what-coalition-will-the-agents-finally-approve?*').

Semiprivate Knowledge

The coalition formation process is given by negotiation capabilities of the coalition leader (that takes care for what can be provided from within the alliance) and team leaders (who suggest service providers form outside of the alliance). In order to understand coalition/team leader decision making (how $\chi^*(m)$ can be constructed) we need to acquire knowledge that the coalition leader stores in its acquaintance model. As explained above, in the acquaintance model there is semi-private knowledge describing agents resources. Provided that the meta-agent can access the copies of agent messages, this task is easy, as the acquaintance models are maintained by means of the inter-alliance communication. The meta-reasoning, for which we use the concept of reasoning **simulation** is described in section 4.1. Given the accessible semi-private knowledge reasoning simulation may provide prediction of the phase

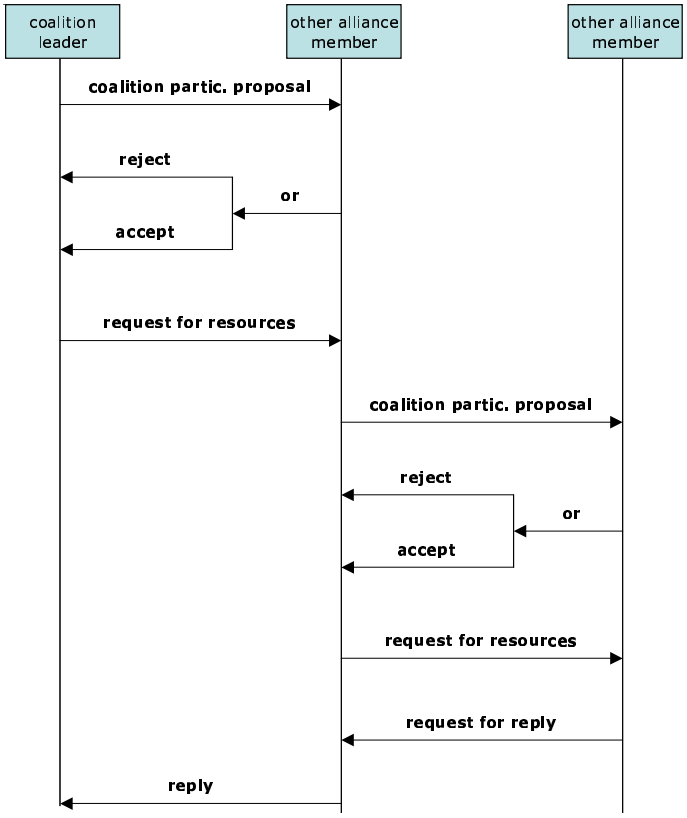

Fig. 1.3. Schema of interagent communication.

1 (Coalition Leader Selection) and phase 2 (In-alliance Coalition Formation) of the coalition formation process.

Private Knowledge

For understanding the decision model of an alliance member (how we can arrive from $\chi^*(m)$ to $\chi(m)$, in other words to know whether the agent will provide the requested services) we need to identify its private knowledge that specifies primarily agents collaboration preferences and restrictions. This piece of knowledge cannot be simply re-constructed from the communication exchange. It need to be identified by the deductive or inductive meta-reasoning process.

In the CPlanT system, every agent A in the community has a decision making algorithm ψ_A which the agent uses to decide whether it will accept the specific team – $\chi^*(m)$:

$$\psi_A(\chi^*(m)) \rightarrow \begin{cases} yes & A \text{ accepts } \chi^*(m) \text{ proposal,} \\ no & A \text{ refuses } \chi^*(m) \text{ proposal.} \end{cases} \tag{1.4}$$

Different agents decide on top of different sets of their private knowledge – $\mathsf{K}_{\mathsf{pr}}(A)$[1]. Agent tries to verify whether the acceptance of participation in given team is consistent with its knowledge (including also public knowledge – $\mathsf{K}_{\mathsf{p}}(A)$):

$$\psi_A(\chi^*(m)) \rightarrow \begin{cases} yes & \iff \mathsf{K}_{\mathsf{p}}(A) \cup \mathsf{K}_{\mathsf{pr}}(A) \vdash accept(\chi^*(m)) \\ no & \iff \mathsf{K}_{\mathsf{p}}(A) \cup \mathsf{K}_{\mathsf{pr}}(A) \nvdash accept(\chi^*(m)) \end{cases} \tag{1.5}$$

In the same manner, we may introduce an abstract notion of the community decision making algorithm that shall predict whether the community of agents θ will be able to fulfill the team request with the mission m:

$$\begin{aligned} &\psi_\theta(m) \rightarrow yes \iff \\ &\quad \exists \chi^*(m) \subseteq \theta \; \forall A \in \chi^*(m) : \mathsf{K}_{\mathsf{p}}(A) \cup \mathsf{K}_{\mathsf{pr}}(A) \vdash accept(\chi^*(m)) \end{aligned} \tag{1.6}$$

Similarly for rejection.

Let us assume that we work with a multi-agent system where agents communicate using FIPA-like communication protocols[2]. Detecting agents' private knowledge is based on observation of the `REQUEST` interaction protocol. An agent replies to a `request` message by an `accept` message if it agrees to provide the requested services for the mission while it replies with a `refuse` message if he does not like to be part of the proposed coalition.

Unlike in the classical `REQUEST` interaction protocol, where in the content of the message, there is the requested service, in CPlanT system, the proposed team is send in the body of the message.

Predicting object agent's decision can be used for e.g. improving efficiency of the collective behavior, reduction of communication traffic or improving the quality of the coalition.

In the sections 4.2 and 4.3, we explore possibilities of usage automated reasoning and machine learning algorithms for the answering whether an agent will eventually accept participation in a coalition and provide requested resources.

[1] In this paper, we suppose, that the set $\mathsf{K}_{\mathsf{pr}}(A)$ is unchangeable during the whole agent's life.

[2] which is the case of CPlanT

2

Abstract Meta-Agent Reasoning Architecture

The central point of the meta-agent's operation is an appropriate model of the community. This model has to be expressed in an appropriate language of adequate granularity. As any other high-level knowledge structure, the model can be represented in implicit way, (e.g. piece of procedural program, characteristic function or set of rules) or in explicit way, logical theory that consists of the true fact that the meta-agent knows about the object agents. The model can be treated in two possible ways. It can be either maintained by

- **deductive reasoning** techniques, when the model contains only formulae that logically follow from the monitored information (e.g. the model will not be in conflict with a future possible event that may happen in the future), or
- **inductive reasoning** techniques that may produce an *approximative model* that includes e.g. generalization formulae, that may prove to be in conflict with a future possible event.

The former type of reasoning produces an exact and 'safe' model of the community. The latter type of reasoning may provide more information while it can misclassify. For the approximative model see section 2.4. In the following we will be explaining the concept of deductive meta-reasoning.

2.1 Model

Let us denote any fact φ, that the meta-agent knows about an agent A by the predicate $\mathsf{bel}_A(\varphi)$. This shall be any formula that contains a predicate(s) with A in its argument. We denote the formula $\mathsf{bel}_\theta(\varphi)$ a fact φ that the meta-agent knows about the community of agents θ[1]. It contains formulae that contain a predicate(s) with any of the agents $A \in \theta$ in its/their argument(s). Let us call either of these formulae a belief-formula. Now, we may define the explicit

[1] θ denotes any sub-community of the whole community of the agents: $\theta \subseteq \Theta$ [7].

model of the community maintained by the meta-agent A^m as a collection of such a belief-formulae:

$$\mathsf{model}(\mu^+(A^m)) = \left\{\varphi |\ \exists\, \theta \subseteq \mu^+(A^m) : \mathsf{bel}_\theta(\varphi)\right\}. \tag{2.1}$$

Generally we have three basic types of belief formulae in the explicit model of the community. For any reasonable manipulation with the model we need to specify the relevant default properties of the object-level community. We will assume these properties to be generally true, to be always true and to be known to all agents before any event in the community happens. Let us call these formulae **background-belief-formulae** and denote them by the predicate $\mathsf{gb}(\varphi)$. The meta-agent A^m is expected to store all formulae these in the knowledge structure referred to as the **background-belief-base** – gbb, formally:

$$\forall \varphi : \mathsf{gb}(\varphi) \Leftrightarrow \varphi \in \mathsf{gbb}. \tag{2.2}$$

In the machine learning domain this knowledge is often referred to as the *background knowledge*. In the most transparent case, this is a theory in first order logic.

Besides, we have facts that were acquired by the meta-agent monitoring capabilities and they very often correspond to the events that happened in the community. Such a fact is referred to as a **event-belief-formulae**. Let us denote such a formula that represents an event happening in the time t by the predicate $\mathsf{eb}_t(\varphi)$ stored in meta-agent's **event-belief-base** – $\mathsf{ebb}_t(\mu^+(A^m))$, formally:

$$\mathsf{ebb}_t(\mu^+(A^m)) = \{\varphi | \exists t_i \leq t : \mathsf{eb}_{t_i}(\varphi)\} \tag{2.3}$$

In the following text, we will denote $\mathtt{event}_t$ a formula φ such a $\mathsf{eb}_t(\varphi)$.

Finally we have the **assumed-belief-formulae** (denoted as $\mathsf{ab}_t(\varphi)$ that are products of the meta-reasoning process facts that happened in the community. In the same way, these are stored in the meta-agent **assumed-belief-base** – $\mathsf{abb_t}(\mu^+(A^m))$, that is formalized similarly to 2.3.

Using the explicit representation we will want at any time instance the meta-agent's model of the object level community to consist of these three knowledge structures:

$$\mathsf{model}_t(\mu^+(A^m)) = \mathsf{gbb} \cup \mathsf{ebb}_t(\mu^+(A^m)) \cup \mathsf{abb}_t(\mu^+(A^m)). \tag{2.4}$$

2.1.1 Model Maintenance

Now, let us briefly mention how the model may be constructed and exploited. The meta-reasoning process in multi-agent system is built upon three mutually interconnected computational processes (figure 2.1):

1. **monitoring** – process that makes sure that the meta-agent knows the most it can get from monitoring the community of object agents. This

process builds the base of the model – model of the community and implements the introspective integrity (in the sense of [10]). assure perfectness of the model.
2. **reasoning** – this process manipulates the model of the community so that true facts (other than monitored) may be revealed. Within the reasoning phase, the meta-agent tries to maintain truthfulness of the model.
3. **community revision** – a mechanism for influencing operation of the object agents' community. This process is inevitable if the meta-agent reasoning phase results in such a goal hypothesis, that is not true in the community, while it ought to be (such as efficiency improvements or agents' awareness of the intruder operating in the system). In this phase, the meta-agent may also affect the operation of the community in order to improve the meta-reasoning process.

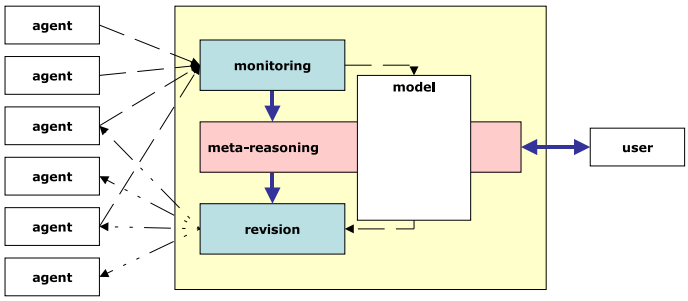

Fig. 2.1. Meta-Reasoning Architecture – A Process View

In this document, we will discuss the reasoning phase in more details. Description of the monitoring phase and the community revision phase can be found in [2].

2.2 Monitoring

The key difference between classical reasoning task and meta-reasoning task in multi-agent systems is that the latter task need to be supported by the monitoring process. The monitoring process enables reasoning in multi-agent systems due to the ability to generate **event belief formulae** from the monitored **observations**. The observation depends on the monitoring approach and possibilities of the object agents community. It can be for example: copy of the communication act between the object agents, information about an object agent's internal state change, etc. The monitoring process then generates the event belief formula in the form suitable for the use of reasoning technology, for example: unit clauses for version space algorithm, logical formulae for automated reasoning or ILP technology, etc.

2.3 Reasoning

Now, we will talk about the reasoning process in more details. The meta-reasoning agent's reasoning operation can be carried out in three different phases:

- **init***time*: initialization time when the meta-agent starts to reason about the system before it receives any event from the community,
- **revise***time*: the instance of the time when an event in the community happens and the community model is automatically revised and
- **inspect***time*: when the user (or other agent possibly) queries the model in order to find out about truthfulness of the goal hypotheses.

Balancing the amount of computational processes in the **revise***time* and the **inspect***time* is really crucial. The proper design depends on the required meta-reasoning functionality. While for visualization and intrusion detection the most of computation is required in the **revise***time*, for explanation, simulation and prediction an important part of computational processes will be carried out in the **inspect***time*.

2.3.1 Community Model Revision

Let us introduce the **community model revision** operator – $\uplus$, that is expected to happen in the **revise***time* exclusively. The community model revision represents the change of the model model_t in the time t with respect to the new formula $\mathtt{event}_t$ that describes an event in the object-level community θ[2]:

$$\mathsf{model}_t(\theta) \uplus \mathtt{event}_t \rightarrow \mathsf{model}_{t+1}(\theta) \tag{2.5}$$

There are different types of events that initiate the community model revision process in the **revise***time*. We talk primarily about initiating a contract-net-protocol, team allocation request, accepting or rejection of the team allocation request, informing about actual resources, etc. (described in appendix A).

In the case of deductive meta-reasoning , we may distinguish between two marginal effects of the community revision operation – $\uplus^{\max}$ and $\uplus^{\min}$ as follows:

$$\mathsf{model}_t(\theta) \uplus^{\max} \mathtt{event}_t = \{\varphi | \mathsf{model}_t(\theta) \cup \{\mathtt{event}_t\} \vdash \varphi\} \tag{2.6}$$

$$\mathsf{model}_t(\theta) \uplus^{\min} \mathtt{event}_t = \mathsf{model}_t(\theta) \cup \{\mathtt{event}_t\}, \tag{2.7}$$

where φ is a formula.

[2] In tho following text: $\theta = \mu^+(A^m)$.

The $\uplus^{\max}$ operator revises the model so that it contains all possible true facts that logically follow from the original model – $\mathsf{model}_t(\theta)$ combined with the new event – $\mathtt{event}_t$. The $\uplus^{\min}$ operator only appends the new formula to the model. In many cases, the $\uplus^{\max}$ operator is hard to achieve as the resulting model may be infinite – we introduce such a model as an abstract marginal concept. The model that results from the $\uplus^{\min}$ model revision shall be always a subset of the model constructed by $\uplus^{\max}$ operator.

When designing the community model revision process, we seek such an operation $\uplus$ that

$$\mathsf{model}_t(\theta) \uplus^{\max} \mathtt{event}_t \supseteq \mathsf{model}_t(\theta) \uplus \mathtt{event}_t \supseteq \mathsf{model}_t(\theta) \uplus^{\min} \mathtt{event}_t \quad (2.8)$$

The closer our operation gets to $\uplus^{\min}$ the faster is the model revision process and more complex should be the computational process in the **inspect***time*. The closer we are to $\uplus^{\max}$ the easier should be the query process while the revision process is getting really complex.

The background belief base $\mathtt{gbb}$ does not change in the time. Only the $\mathsf{ebb}_t(\theta)$ changes if the community revision operator $\uplus^{\min}$ is used, while both $\mathsf{ebb}_t(\theta)$ and $\mathsf{abb}_t(\theta)$ change in the time if another community revision operator is used.

The concept of model revision is closely related to the concept of weak and strong update in the knowledge engineering area [11].

2.3.2 Community Model Inspection

During the **inspect***time* (in any time t), the computational process of **community model inspection** provides the user (or any other agent) the reply for a queried question – goal formula ($\mathtt{goal}_t$). We introduce an operator $\nsim$ for model inspection, which replies as follows:

$$\mathsf{model}_t(\theta) \nsim \mathtt{goal}_t \rightarrow \begin{cases} \mathit{yes} & \text{if } \mathtt{goal}_t \text{ is provable in } \mathsf{model}_t(\theta), \\ \mathit{no} & \text{if } \neg\mathtt{goal}_t \text{ is provable in } \mathsf{model}_t(\theta), \\ \mathit{unsure} & \text{otherwise.} \end{cases} \quad (2.9)$$

If the considered goal formula contains existentially quantified variables, the reply can also contain their possible substitution.

In the case of deductive meta-reasoning , the **minimal** version of the community model inspection process corresponds directly to checking occurrence of the goal formula within the model. The relevant formula can be retrieved from the model with no further reasoning.[3]

[3] The goal formula can contain existentially quantified variables that get bound to constants during the instantiation process.

$$\mathsf{model}_t(\theta) \mathrel{\sim\!\!\!\!\!\!\rightarrow}^{\min} \mathtt{goal} \rightarrow \begin{cases} \mathit{yes} & \text{if } \mathtt{goal} \in \mathsf{model}_t(\theta), \\ \mathit{no} & \text{if } \neg\mathtt{goal} \in \mathsf{model}_t(\theta), \\ \mathit{unsure} & \text{otherwise.} \end{cases} \tag{2.10}$$

In order to use the minimal version of the model inspection in the **inspect***time*, we need the maximal (or close to maximal) model revision ($\uplus^{\max}$) in the **revise***time* of the meta-reasoning life-cycle.

If the reasoning triggered by the event (in **revise***time*) has not produced the queried formula, the inspection process will be a more complex operation than simply parsing the existing model. The meta-agent is expected to employ reasoning in order to find out whether the requested goal formula logically follows from the model:

$$\mathsf{model}_t(\theta) \mathrel{\sim\!\!\!\!\!\!\rightarrow} \mathtt{goal} \rightarrow \begin{cases} \mathit{yes} & \text{if } \mathsf{model}_t(\theta) \vdash \mathtt{goal} \\ \mathit{no} & \text{if } \mathsf{model}_t(\theta) \vdash \neg\mathtt{goal} \\ \mathit{unsure} & \text{otherwise.} \end{cases} \tag{2.11}$$

2.4 Approximate Model of the Community

Besides representing the model of the object agent community as a logical theory it can be also represented by an appropriate approximation ψ_A^* of the object agent's decision algorithm ψ_A.

The problem of revealing ψ_A^* can be transformed to the problem of learning unknown function, e.g. in the words of machine learning. Every event $\mathtt{event}_t$ is decomposed into a pair containing the request for specific service τ_t and object agent's reactions: $\mathtt{event}_t = [\tau_t, \psi(\tau_t)]$, where τ_t also specifies question that can be answered by learned ψ_A^* algorithm. Therefore, we use the history of all prior events stored in the event belief base as a *training set*. It is stored in the following format:

$$\mathsf{ebb}_\mathsf{t} = \{[\tau_i, \psi(\tau_i)]\}_{0 \le i \le t} \tag{2.12}$$

The ebb_t event belief base and gbb background belief base are stored in constructed model model_t.

2.5 Properties of the Model

Let the community model be defined as in 2.1. Suppose, that it is possible to decompose the model $\mathsf{model}(\theta)$ of the community θ to the models of singleton agents $A_i \in \theta$. We say that such model is **decomposable**:

$$\mathsf{model}(\theta) = \bigcup_{A_i \in \theta} \mathsf{model}(\{A_i\}), \tag{2.13}$$

where $\mathsf{model}(A_i)$ denotes a meta-reasoning model of the singular agent A_i. If such a decomposition is not possible, we call the model **undecomposable**. This is the case of emergent behavior in the multi-agent systems, where there is no single agent that would approve participation in the mission itself, while collectively the group can do it.

In both cases, the models contain the same information, but every type is suited to a different task of meta-reasoning.

2.5.1 Decomposable models.

A decomposable model can be created as a union of object-agents' models. Then, there can be directly represented information about singleton agent's decision models. While the information about the community as whole can be acquired only by the simulation of all object agents. Decomposable model can be effectively used for prediction and explanation of singleton agent's reaction.

2.5.2 Undecomposable models.

An undecomposable model contains directly the information about the community, while the information about agent's decision making algorithm is represented only implicitly – in some cases, it can be extracted by a restriction of the model (this case occurs also in our domain). The model is undecomposable if it contains any knowledge about emergent behavior of object community, i.e. behavior, that is not peculiar to any member of the community. Undecomposable model can be used to find out more general information about the community of object agents. It can be used e.g. for visualization.

Example 1 Decomposable and undecomposable models.

We have a task to reason about a neural network (as an abstraction of a set of object-level agents). If we want to create a decomposable model, we should monitor every singleton neuron and find out how does it work. But this knowledge will not say us how the whole neural network is evaluating a given input. We can only simulate all neurons to get the output value. Creating an undecomposable model, we monitor only the inputs and outputs of the whole network. Created model can be able to evaluate given inputs, but does not have to contain any knowledge about singleton neurons.

While the version space algorithm (see section 4.3.1) works with decomposable models only, automated reasoning (section 4.2) and ILP (section 4.3.2) work with both types of model.

2.5.3 The concept of $\mathsf{model}^{\mathsf{T}}(\theta)$ and $\mathsf{model}^{\mathsf{G}}(\theta)$

We shall distinguish between two basic types of model updating: $\uplus^{\mathsf{T}}$ and $\uplus^{\mathsf{G}}$. The $\uplus^{\mathsf{T}}$ tries to generate all true facts about the object-level community and

stores them in the model $\mathsf{model}^{\mathsf{T}}(\theta)$. Then, when the goal formula is asked, the $\sim\!\!\!\vdash$ operator tries to find out whether this formula or its negation is implied by the $\mathsf{model}^{\mathsf{T}}(\theta)$ model. If it is, the reply is *yes* or *no*. Otherwise, the reply is *unsure*.

The $\uplus^{\mathsf{G}}$ tries to find out more general rule which distinguishes positive and negative events. This rule is then evaluated for a given `goal` when the $\sim\!\!\!\vdash$ operator is used. This rule can be found out by e.g. a machine learning algorithm.

There is one substantial difference between these two approaches. All created facts in the $\mathsf{model}^{\mathsf{T}}(\theta)$ are valid and they stay valid even as new events come. We can say that the model is monotonic in time (we suppose monotonic behavior of object agent). While, the $\mathsf{model}^{\mathsf{G}}(\theta)$ can change the evaluation of goals that have not been in ebb. The advantage of this approach is that we do not need any information about the object agent's decision making algorithm, even thought this information can significantly improve the quality of created model.

Both operators maintain consistency of created model with the processed events and background theory. First approach also ensures the correctness (it will always respond correctly or *unsure*)[4]:

$$\begin{array}{lll}
\mathsf{model}^{\mathsf{T}}(\theta) \sim\!\!\!\vdash \mathtt{goal} \rightarrow \mathit{yes} & \Rightarrow \mathsf{model}^{\mathsf{G}}(\theta) \sim\!\!\!\vdash \mathtt{goal} \rightarrow \mathit{yes}, & \\
\mathsf{model}^{\mathsf{T}}(\theta) \sim\!\!\!\vdash \mathtt{goal} \rightarrow \mathit{unsure} & \Rightarrow \mathsf{model}^{\mathsf{G}}(\theta) \sim\!\!\!\vdash \mathtt{goal} \rightarrow \begin{cases} \mathit{yes}, \text{ or} \\ \mathit{no}, \text{ or} \\ \mathit{unsure}, \end{cases} & (2.14) \\
\mathsf{model}^{\mathsf{T}}(\theta) \sim\!\!\!\vdash \mathtt{goal} \rightarrow \mathit{no} & \Rightarrow \mathsf{model}^{\mathsf{G}}(\theta) \sim\!\!\!\vdash \mathtt{goal} \rightarrow \mathit{no}, &
\end{array}$$

where

$$\begin{aligned}
\mathsf{model}^{\mathsf{T}}(\theta) &= \mathsf{model}_0 \uplus^{\mathsf{T}} \mathsf{ebb}, \\
\mathsf{model}^{\mathsf{G}}(\theta) &= \mathsf{model}_0 \uplus^{\mathsf{G}} \mathsf{ebb}.
\end{aligned}$$

In this work, the $\uplus^{\mathsf{T}}$ operator is represented by the resolution principle (section 4.2) and the $\uplus^{\mathsf{G}}$ operator is implemented using inductive logic programming (section 4.3.2). The version space algorithm (section 4.3.1) creates hypotheses in the form of $\mathsf{model}^{\mathsf{G}}(\theta)$, but by working with all possible hypotheses it implements the $\uplus^{\mathsf{T}}$ operator.

[4] All knowledge structures and operations are assumed here to be true and performed at the same time. The t index has been omitted here for simplicity.

3

Meta-Reasoning in CPlanT: Monitoring

The monitoring process enables meta-reasoning activities in multi-agent systems with generating of event belief formulae. The event belief formulae can be generated from the copies of the messages, from the observation etc. We distinguish between two basic environments:

- **collaborative environment** – the object agents are aware that they are monitored and they support it and
- **non-collaborative environment** – the object agents do not want to be monitored and they do not provide the meta-agent with any support.

We focused on both types: collaborative and non-collaborative environment. There are technologies appropriate for the monitoring process in collaborative environment [12]. However, we have suggested a novel technology (based on the intruder agents) for the monitoring process in non-collaborative environment. The intruder agent acts as an ordinary object agent. It observes the environment and it generates event belief formulae. We have designed collaboration mechanisms and social commitments between the intruders. Meta-reasoning process will be improved in the sense of the number of generated event belief formulae. We have designed a proactive approach to monitoring where the meta-agent is able (via intruders) to affect the operation of the multi-agent system and the meta-reasoning process in general.

3.1 Short Review of Approaches to Monitoring

This section has been integrated already in the report [7]. For the sake of clarity and completeness of this report we have decided to repeat the review in this place.

Previously, people investigated ways of monitoring teams and communities of either competing or collaborative agents [13]. In principle, we distinguish between the query-based and subscription-based monitoring [12]:

- **query-based monitoring**, when the meta-reasoning agent itself is trying to detect new information or inspect validity of the already stored knowledge. The query-based (active) monitoring can be implemented by *communication* where the meta-agent periodically checks with the object agents whether the monitored information has been changed. This has been done previously by *periodical revisions* [14], [15]. Similar option is implicit monitoring via *environment* where the agents act and interact. Huber and Kaminka [16], [13] suggest active monitoring via *plan-recognition*.
- **subscription-based monitoring**, when the meta-reasoning agent gets notified when truthfulness of the monitored proposition changes [17]. This type of monitoring is usually implemented by the *subscribe-inform* conversation protocol. The subscriber queries/subscribes an object agent for information that describes its computational state, beliefs, or goals. The object agent replies and keeps informing the subscriber each time this information becomes invalid. In [13] there was introduced the *monitoring selectivity problem* i.e. the challenging problem of deciding how much of monitoring is relevant and necessary for performing the required reflective task.

Naturally, the most appropriate monitoring/meta-reasoning component of a multi-agent system would be its central communication element (agent). Here the information can be collected and analyzed. From many important reasons such as robustness and fault tolerance of the system, autonomy of agents, assuring information privacy, dynamics and flexibility of the system, we wanted to avoid such a centralized approach.

If the object agents act autonomously and communicate peer-to-peer, a meta-reasoning agent may monitor the communication traffic by observing the communication exchange among the object agents. It subscribes the object agent for copies of communicated messages. This philosophy distinguishes the central communication agent from the meta-agent. Had the former one failed to operate, the entire community is paralyzed and can not continue to operate, while the operation of the latter one is independent from functioning of the community of object agents.

A more complicated problem, which has not been studied thoroughly yet, is monitoring of the object agents that do not want to be monitored (in the case of non-collaborative meta-reasoning). Another challenging problem is monitoring a community of agents in distributed manner. Several monitoring agents may be in charge of monitoring different parts of the community, in different times or monitoring different aspects of object agents' operation.

The monitoring process has been investigated in the past. The object agent's state reconstruction is performed in [18]. The domain is characterized by the huge amount of the communication acts, which are replaced by probability propagation over the possible states and restricted by the team oriented program. The socially attentive monitoring for failures identification in inter-agent communication has been reported in [13]. We will exploit a similar

approach in non-collaborative environment, where we will identify the object agent's behavior patterns from the commitments between the object agents. The main difference between our domain and the socially attentive monitoring domain is in the amount of the communication acts. Our domain is characterized by a limited amount of communication acts with the information-rich content of the messages.

3.2 Monitoring in Collaborative Environment

In collaborative environment the object agents support the meta-reasoning process. They can be informed about the meta-reasoning results and to use new knowledge for efficiency improvement of their decision making process [19]. This is the reason why they support it. We have designed and implemented the *reasoning simulation* (see section 4.1) technique for deducing the event belief formulae from the copies of their messages.

3.2.1 Subscription-based Monitoring

The meta-agent subscribes the object agents for informing about any changes of monitored state of the object agent [12]. In our case, the meta-agent subscribes the copies of the messages between the object agents [19].

The meta-agent subscribes the object agents for copies of the messages via a mask. The mask is defined with specific performative, ontology, language and performative of the message content. The mask defines if the message is sent or received by the object agent. The meta-agent subscribes the object agent via several masks. The subscribed object agent compares sent/received message with a set of the masks. If any mask matches the copy of the message, it is forwarded to the meta-agent.

Example 2 The mask definition.

The mask sent to the object agent "Suffer Terra Government" can be defined in a such way:

`Sender`: X
`Receiver`: "Suffer Terra Government"
`Performative`: request
`Ontology`: cplant
`Language`: kif

Now the object agent "Suffer Terra Government" forwards any received message with performative "request", ontology "cplant" and language "kif" to the meta-agent.

3.2.2 Generating Event-Belief Formulae

The event belief formulae for the meta-reasoning process are generated from the copies of the object agents' communication acts. There can appear two types of the messages:

- these, from which we can directly generate the event belief formulae without further analysis and
- these, which have to be submitted to a reasoning simulation process in order to generate an event belief formulae.

The event belief formulae generation from the second type of the messages is more complicated, it is described in section 4.1.

There are two situations when event belief formula is generated without further analysis:

- the object agent has accepted the team allocation request – there are generated the event belief formulae informing about it and that the team leader accepts the number of the team members equivalent to the team proposed within the task in the team allocation request,
- the object agent has refused the team allocation request – there are generated the event belief formulae informing about it and that the team leader accepts the number of the team members equivalent to the team proposed within the task in the team allocation request.

See section B for the technical description of the monitoring process in collaborative environment.

3.3 Monitoring in Non-collaborative Environment

There are several applications of the multi-agent systems where the agents own private knowledge [1]. They try to keep the private knowledge then they do not want to be monitored to disable reasoning about them. We suggest a specific approach how to reconstruct the information about the object agents, provided they do not support the meta-reasoning activity. This approach is based on creation of a *fake agent*, that besides an ordinary agent's functionality is capable of triggering a required activity of the community and it has an access to the resulting event belief formulae. We have proposed and designed a special type of the agent **intruder agent** for the monitoring process in non-collaborative environment. In the community, there operate several intruders, they are aware of each other and cooperate. The intruder agent can take two roles:

- it observes the environment and it generates the event belief formulae and
- it generates stimulation events [2] to the community of the object agents in order to initiate requested behavior and acquire the corresponding event belief formulae (see proactive monitoring in section 3.4).

3.3.1 Monitoring with Intruders

Intruder registers with the community and joins an alliance. If it succeeds it has an access to the object-agent's semiprivate knowledge and it is able to generate the event belief formulae about the alliance members. The event belief formulae are forwarded to the meta-agent for further analysis. Intruders act as a ordinary object agent. They provide the alliance with services in coalition planning process but they take the role neither of the team leader nor of the coalition leader. They are passive, they observe only. The meta-agent controls them.

Community Coverage

We say that the intruder agent A_j^i **covers** the object agent if the object agent is in the monitoring scope of the intruder agent – $A_i \in \mu^+(A_j^i)$. Please note, that the property defined in (1.1) is not valid in the non-collaborative environment and thus we do not require the monitored agent to monitor the intruder. We say that the object agent A_i is **covered** only if there is any intruder agent, which covers the object agent – $\exists A_j^i : A_i \in \mu^+(A_j^i)$. The community of the object agents is **fully covered** only if all the object agents are covered.

The important notion is the **degree of the coverage**. It depends on the amount of generated event belief formulae and deduced knowledge within the **revise***time* and the **inspect***time*. This concept has not been fully elaborated until now.

Intruders and meta-agent are aware of the community coverage. The meta agent has to process the situation when a new object agent appears in the community. It waits for a specific time limit if the new object agent will be covered by any existing intruder, otherwise the meta-agent creates a new intruder (see below). The intruder agent, which covers an object agent who has just deregistered, informs other intruders and the meta-agent. The intruder deregisters from the community if it covers no object agent.

Activation of New Intruder

The meta agent assigns to the intruder the parameters of the object agent which should be covered – monitored object agent (city, country, type). The meta agent knows the monitored object agent's parameters because it is a public knowledge accessible within whole multi-agent system [1]. It assigns to the intruder available services, meta-agent's name and names of other existing intruders. The new intruder agent finds the meta-agent, existing intruders and the monitored object agent within the registration phase.

When the intruder finishes the registration process, it continues with an alliance forming process. The intruder negotiates with the monitored object agent. If the negotiation succeeds (the monitored object agent would accept

the intruder as a new alliance member) the intruder receives alliance members of the monitored object agent. The intruder continues to negotiate step by step with other alliance members. If any object agent refuses the request for the alliance participation the intruder agent informs the meta-agent that alliance formation process was unsuccessful.

The meta-agent assigns new parameters to the intruder agent if the alliance formation process was unsuccessful. The new parameters are determined by the monitored agent's alliance members. The meta-agent has to wait for a new monitored object agent's alliance member if there either is no alliance member now or parameters of all monitored abject agent's alliance members were tested. It is possible that the intersection of the alliance restrictions of the object agents in one alliance is empty set. Such an alliance has to be monitored by proactive monitoring (see section 3.4).

The intruder informs the meta-agent and other intruders when it joins the monitored object agent's alliance. It subscribes the object agents for their services and it is prepared for the participation in planning phases and for event belief formulae generation.

3.3.2 Generating Event Belief Formulae in Non-collaborative Environment

The intruder is an ordinary alliance member. It participates in team allocation process and with services in resource allocation process. There are two results of the team allocation process: the team has been allocated successfully (the resource allocation process has already started) and the team has not been allocated because any member has refused the task. There are two possible event belief formulae that may be generated when the team has been allocated:

- all the team members except the team leader accept the task and
- the team leader accepts the number of the team members equivalent to the team proposed within the task in the team allocation request.

The event belief formulae are generated when the team allocation process failed:

- one of the team members besides the team leader does not accept the task proposed within the team allocation request and
- the team leader accepts the number of the team members equivalent to the team proposed within the task in the team allocation request.

3.3.3 Collaborative Monitoring

The object agents adopt their decision making algorithm with respect to unsuccessful attempts to allocate a team. The intruders would not be able to detect team rejections if they were not present in the system when the team has been allocated unsuccessfully. The intruders are not able to generate event

belief formulae about the team leaders' restrictions, because the team leader does not contract itself when it allocates the team. These reasons are why the intruders have to cooperate. They have to share allocated services within the teams in order to detect similar situations. Then the intrudes could generate event belief formulae when these situations appear:

- any object agent could participate with the services within the team but it has not been included in the team,
- the team leader could participate with the services but it did not.

The monitoring efficiency should be improved in the sense of the number of generated event belief formulae. This is due to the ability to generate event belief formulae about the team leader and about the team rejections from the time, when the intruders did not participate in the community.

3.3.4 Socially-Attentive Monitoring

The intruder does need to be included in the team. It should be able to reconstruct allocated team even in the case that the intruder does not participate. We were inspired by [13] where the social commitments are used for improving of the monitoring efficiency. The intruder is able to detect the team members without its participation due to the social commitments (subscribe/advertise mechanism) cross monitored alliance. Due to the socially attentive and collaborative monitoring the intruders can generate not fully instantiated event belief formulae even in the cases when they participate in no team. The monitoring efficiency should be improved in the sense of the number of generated event belief formulae. This is due to the ability to reconstruct the teams even in the case that not all intruders participate within the team.

3.4 Proactive Monitoring

Very often the community of object agents performs so little activity or a pattern of activity that is little relevant to the course of desired decision making process. In such a case the meta-agent needs to wait until required information will be available. Alternatively, the meta-agent may want to initiate such an activity within the community that the desired event happens. The meta-agent need to know:

- which is the desired event
- how this event can be initiated

For answering the first question we suggest to use the concept of **abductive reasoning** that has been successfully used in the domain of expert systems in the past. In the collaborative environment the meta-agent may simple send a `REQUEST` message to the agent in question, in order the event to happen.

However, in the adversarial domain, this is rather more difficult and we need to use e.g. the intruders who interact with the ordinary object agents.

Proactive monitoring is not covered by this project report.

4

Meta-Reasoning in CPlanT: Reasoning

Given the right knowledge structure describing the object level community – model, the reasoning process within meta-reasoning is given by two reasoning operators: (i) the community revision operator $\uplus$ and (ii) the community inspection operator $\upharpoonleft\!\!\rightarrow$. Meta-agent based on this abstract architecture has been implemented for agent's private knowledge detection in the CPlanT multi-agent system. We have implemented three distinct reasoning methods that represent completely different approaches to AI reasoning:

- **Deduction:** where we use explicit representation of true facts in the community and techniques of theorem proving and automated reasoning as reasoning operators.
- **Induction:** where we construct and administer hypothesis about singular agent's decision rule and we use machine learning algorithms to revise and inspect the model.
- **Simulation:** where we simulate coalition/team leaders decision making model Both methods use the same monitoring process.

While deduction and induction is used for private knowledge detection, the simulation mechanism processes the semiprivate knowledge and tries to predict a coalition proposal that would be suggested by the respective coalition/team leader.

4.1 Simulation

The object agent may have two reasons why it refuses the mission within the team request from the coalition leader:

- the mission parameters do not comply with its private knowledge describing collaborative behavior preferences and restrictions, or
- neither the agent himself nor its alliance member has available resources to provide the coalition with.

We monitor the object agents' semi-private knowledge and the object agent's acquaintance model to be able to distinguish these two situations and in order to improve the monitoring process in the sense of the number of generated event belief formulae. We are able to generate more event belief formulae if we simulate the object agent's decision making algorithm that it uses for the contract net protocol negotiation, which is before the team allocation process [1].

The meta-agent stores an acquaintance model (see 1.4) that includes a yellow-page list of all object agents in the community. In additional, the meta-agent stores copies of the acquaintance models of the object agents (only in collaborative environment). It keeps the object agents' acquaintance models by means of tracking the social commitments between the object agents within alliances (subscribe/advertise mechanism) [1]. The meta-agent can simulate the object agent's decision making process when the object agent receives the team request within the contract net protocol. In order to reconstruct these, the meta-agent needs to know the same acquaintance model which would be used by the object agent. Then the meta-agent produces event belief formulae and it is able to deduce knowledge about the respective agent. See figure 4.1 for reasoning simulation scheme.

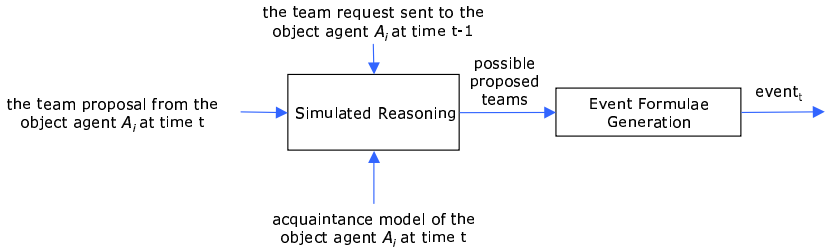

Fig. 4.1. Reasoning simulation.

The team request to the object agent A_i (query message – requested services), the team proposal (inform message - proposed services by the object agent's A_i alliance) and the object agent's A_i acquaintance model are submitted to the reasoning simulation process. The search algorithm finds all possible proposed teams by the object agent A_i. Then event belief formulae can be generated in any case if:

- all possible proposed teams cover any object agent (the requested agent would accept the requestor as a team leader in gbb),
- all possible proposed teams cover the requested agent A_i (the requested agent would accept the requestor as a team leader and it would participate with services in the task location in gbb),

- no possible proposed team cover the requested agent A_i (the requested agent does not accept the requestor as a team leader or it would not participate with services in the task location or both in gbb),
- the requested agent A_i accepts the team counting minimal number of the team members in possible proposed teams.

We suppose that the meta-agent is present in the community before any planning phase of the system. The condition is satisfiable in collaborative environment. The object agents adopt their decision making algorithm with respect to unsuccessful attempts to allocate a team. See an example 3:

Example 3 Agent's adaptive behavior.

If the object agent refuses the task proposed by the team leader within the team allocation request then the team leader will not try to form the team defined in previous task in the location define in the task again.

We can use the reasoning simulation for generation of event belief formulae about the previous task rejections, although the meta-agent has lost some planning actions. The meta-agent can find the object agents which have not been included in possible proposed teams but they have services not covered by the possible proposed teams. For technical and formal specification of the reasoning simulation process see Appendix B.

4.2 Deduction

Prior to discussing the meta-reasoning application, let us give a brief account on the selected theorem proving techniques available.

Theorem proving and automated reasoning covers an important part of the traditional symbolic artificial intelligence, where the essence of building artificially intelligent systems is rooted in manipulation with the symbolic representation of the environment – the object level multi-agent system, in our case. The valid facts about the object agents are represented by means of logical formulae and syntactic manipulation with these formulae is given by the rules of logical deduction.

Resolution Principle

Probably the most popular calculus used in the implementation of reasoning programs is based on the resolution principle [20]. All the logical formulae, that describe the model $\mathsf{model}_t(\theta)$, are supposed to be encoded as a conjunction of clauses, where each is a disjunction of literals – CNF (conjunctive normal form). The negation of the general closure of the goal hypothesis that is about to be proved is then appended to the model. Such a theory undergoes the

process of resolution. When resolving the theory the clauses are put together so that new true clauses get constructed (for instance: from $(\psi \vee \varphi)$ $(\neg\psi \vee \rho)$ is constructed $(\varphi \vee \rho)$) with the ultimate goal to construct the empty clause representing the contradiction. Had there been the contradiction found, the goal hypothesis is proved. Classical rules of mathematical logics such as modus ponens, de-Morgan rules or unification are part of the resolution process. The unification process [20] determines a set of possible substitutions of unbound variables when resolving clauses.

Binary Resolution

The *binary resolution* is a basic resolution rule, which from the two resolvents generates a new resolvent (see [21]). The resolution step is performed over positive literal placed in the first resolvent and negative literal placed in the second resolvent provided that they can be unified. There are number of resolution rules, e.g. *UR-resolution* [21] applies on selected resolvent step by step several unit clauses with an attempt to generate a new unit clause. *Hyperresolution* [22] combines several resolution steps with an attempt to reduce the number of generated resolvents.

Searching Strategies

Selection of the appropriate searching strategy (for selecting resolvents to be expanded) is an important aspect for the success of the theorem proving process. It determines which clauses to make ready for the state-space expansion via generation of the new clauses due to the resolution prinicple. Searching strategy controls the whole of proving process. The searching strategies in theorem proving are similar to the searching strategies used in problem solving algorithms [23]. Let us mention the following searching strategies. The *breadth first searching strategy* generates first the resolvents with derivation depth equal to one, then equal to 2 and so on. The *linear* resolution [24] always resolves a clause with the most recently derived resolvent.

Set of Support Strategy: For reasons explained below we have favored the set of support strategy [25], that is one of the most powerful resolution strategies. It tries to find a inconsistency in the set of axioms from the inconsistency justification. The resolvent is generated if at least one parent is supported by the goal hypothesis.

There are several heuristic ways how the resolution process may be optimized. We have made a good use of the following:

- **Subsumption:** A clause C *subsumes* a clause D if and only if there is a substitution σ such that $C\ \sigma \subseteq D$. D is called a *subsumed* clause [26]. The subsumption process reduces the space of possible resolvents by removing more specific clauses (like D) if there is a more general clause present in the theory (like C). The *forward subsumption* process attempts to refute a

new resolvent with respect to already generated resolvents. The *backward subsumption* process tries to make already generated resolvents as non-available for inference process with respect to the new generated resolvent.

- **Model Elimination:** This variation of the resolution strategy [27] introduces instead of clauses literal lists (chains) as basic reasoning entities. We have literal lists of two types: A-literal and B-literal. The bracketed A-literals are all present eliminated literals, B-literals are these, which have to be eliminated in order to full eliminate A-literal. The expansion process based on resolution principle expands the rightmost A-literal with the candidates. The reduction process eliminates A,B-literals from the chain under specific conditions. Model elimination can remove a sentence by showing that it is false in some model of the axioms (see below). You can find several theorem provers based on this technology (METHEOR [28], SETHEO [29]).

The *deletion strategy* is the deletion of any tautology and any subsumed clause whenever possible. As we have to worry about computational efficiency in automated reasoning non-complete strategies have to be considered. The unit resolution and the input resolution [20] are two refinements of the linear resolution. The *unit* resolution is restricted to clauses where at least one parent is unit clause. The *input* resolution is restricted to clauses where at least one parent is from the input set of the axioms.

In the following, we will study the automated reasoning and theorem proving operation performed within the **revise***time* the **inspect***time* and application of the concepts lemmatizing a caching within the theorem proving.

4.2.1 Automated Reasoning in Revision Time

We have experimented with different model revision operators, all based on the suggested novel heuristic strategy – referred to as *shortening strategy*. This strategy generates a new clause only if the length of the new clause (here the length equals to the number of the symbols in the clause) is smaller than one of its parents clauses. We have used the strategy within the community revision phase only.

The computational process in the revision time is based on a combination of the shortening strategy (implementing the $\uplus^S$, $\uplus^{\min} \subset \uplus^S \subset \uplus^{\max}$) and simple appending the event belief formula to the model (implementing the $\uplus^{\min}$). We will refer to this process as $\uplus^N$ operator, where N determines the amount of clauses in the initial model suggested to the operation $\uplus^S$ (see eqn. 4.2). We prefer the $\uplus^S$ community revision operate to the $\uplus^{\max}$ abstract community revision operator with respect to the number of generated clauses within the **revise***time* and to an attempt to deduce the unit clauses. $\uplus^S$ operator is not a full equivalent substitution of the $\uplus^{\max}$ operator, however the shortening strategy reflects the behavior of $\uplus^{\max}$ operator. Forward and backward subsumption techniques [26] are included within the community revision process.

The operator $\uplus^N$ has been implemented as follows. Let us split the initial model into two disjunctive parts, denoted as $\mathsf{model}_0^{\min}(\theta)$ and $\mathsf{model}_0^S(\theta)$:

$$\forall\varphi \in \mathsf{model}_0(\theta) : \varphi \in \mathsf{model}_0^{\min}(\theta) \ \mathsf{xor} \ \varphi \in \mathsf{model}_0^S(\theta). \tag{4.1}$$

The operation $\uplus^{\min}$ is applied on the first part of the model $\mathsf{model}_t^{\min}(\theta)$ and the operation $\uplus^S$ is applied on the second part of the initial model $\mathsf{model}_t^S(\theta)$, formally:

$$[\mathsf{model}_t^{\min}(\theta) \uplus^{\min} \mathtt{event}_t] \cup [\mathsf{model}_t^S(\theta) \uplus^S \mathtt{event}_t] \rightarrow \mathsf{model}_{t+1}(\theta) \tag{4.2}$$

Let us briefly describe creation of $\mathsf{model}_0^S(\theta)$. It is practically impossible to use the entire initial model for the $\uplus^S$ operation ($\mathsf{model}_0^S(\theta) = \mathsf{model}_0(\theta)$) because then the **revise***time* would become the bottleneck of the whole meta-reasoning process. For the proper partitioning let us assume that

- the set of expected events that can happen in the community (all event belief formulae) is finite and
- there is a specified cap on the maximal admissible depth of the clause derivation within the **revise***time* operation.

All reachable unit clauses (contains neither unit event belief formulae nor their negation) are found by forward propagation from gbb through all expected event belief formulae and the shortening strategy (see the program 1 used for reachable unit clauses constitution, program is invoked with these parameters: gbb converted to CNF form, all expected event belief formulae and maximal admissible depth of the clause derivation, for definition of SOS list see section 4.2.3). Required $\mathsf{model}_0^S(\theta)$ is constituted by backward propagation from all reachable unit clauses. It consists of the parent clauses of all reachable unit clauses from gbb. Then $\mathsf{model}_0^S(\theta)$ is copied to $\mathsf{abb}_0(\theta)$. The process is illustrated by example 4.

The parameter N specifies the maximal admissible depth of the clause derivation within the **revise***time* and then the community revision operator $\uplus^N$. It determines also the size of the $\mathsf{model}_0^S(\theta)$. It controls meta-reasoning's complexity within the **revise***time*. With increasing admissible depth of the clause derivation within the **revise***time* there is increased complexity of the community revision process.

In realistic examples, we may hardly ever enumerate all possible occurrences that may happen in the object level communities. Even if this is possible, we have to face the scalability problem. With an increasing complexity of the community, the partitioning problem is really hard to be solved. However, we must bear in mind that this computational process is to be carried out only in the **init***time* and does not bring much of a computational burden in the **revise***time*.

There are other ways how the $\mathsf{model}_0^S(\theta)$ can be created. We experimented with the random setting that includes the formulae from which it is not possible to deduce any unit clauses. It is not effective for the reasoning process

Example 4 Initial model splitting.

Let us demonstrate above described process on a simple example. Ground belief base gbb is defined (variables are in uppercase, constants in lowercase):

1. $\forall X \forall Y \forall Z\ accept(X,Y,Z) \Leftrightarrow accept\_leader(X,Y) \wedge accept\_city(X,Z)$,
2. $\forall X \forall Y\ \neg accept\_leader(X,Y) \wedge proposal(Y,X) \Rightarrow refuse\_proposal(X,Y,M)$.

We transform background knowledge to CNF form:

1. $\neg accept(X,Y,Z) \vee accept\_leader(X,Y)$,
2. $\neg accept(X,Y,Z) \vee accept\_city(X,Z)$,
3. $accept(X,Y,Z) \vee \neg accept\_leader(X,Y) \vee \neg accept\_city(X,Z)$,
4. $accept\_leader(X,Y) \vee \neg proposal(Y,X) \vee refuse\_proposal(X,Y,M)$,

There are three expected event belief formulae $accept(X,Y,Z)$, $\neg accept(X,Y,Z)$ and $proposal(Y,X)$. Be aware of the fact that the events $accept(X,Y,Z)$ and $\neg accept(X,Y,Z)$ make the theory inconsistent. There can appear empty clauses within the simulation process. We filter them out. Maximal admissible depth of the clause derivation within the **revise***time* equals to 2. There are four reachable unit clauses: $accept\_leader(X,Y)$, $\neg accept\_leader(X,Y)$, $accept\_city(X,Z)$ and $\neg accept\_city(X,Z)$. The literal $refuse\_proposal$ is reachable with maximal admissible depth equals to 3. We use backward propagation in order to find parents of reachable unit clauses, then we can define:

$$\begin{aligned}
\mathsf{model}_0^S(m) = \{\ & \neg accept(X,Y,Z) \vee accept_leader(X,Y), \\
& \neg accept(X,Y,Z) \vee accept_city(X,Z), \\
& accept(X,Y,Z) \vee \neg accept_leader(X,Y) \vee \neg accept_city(X,Z)\}, \\
\mathsf{model}_0^{\min}(m) = \{\ & accept_leader(X,Y) \vee \neg proposal(Y,X) \\
& \vee refuse_proposal(X,Y,M)\}.
\end{aligned}$$

based on the shortening strategy. We suppose that all formulae from $\mathsf{model}_0^S(\theta)$ are always revised within the **revise***time*. It is possible to change it with respect to $\mathsf{ebb}_t(\theta)$ in order to improve efficiency of the community revision phase and then whole meta-reasoning process. The ways how to set $\mathsf{model}_0^S(\theta)$ and how to change $\mathsf{model}_0^S(\theta)$ with respect to $\mathsf{ebb}_t(\theta)$ will be in scope of our further research.

Let us denote $\mathsf{event\_lemmas}_t(\theta)$ as a set of all unit clauses deduced within the **revise***time* until time t. They are called as event lemmas. They will be used for the reduction of the time responses within the **inspect***time*.

See program description n.1 when a new $\mathtt{event}_t$ appears. The program receives an actual content of assumed belief base $\mathsf{abb}_{t-1}(\theta)$ (denoted as abb{t-1} in the program description), the new event $\mathtt{event}_t$ (denoted as event{t}) and maximal admissible depth of the clause derivation. New content of assumed belief base $\mathsf{abb}_t(\theta)$ (denoted as abb{t}) and new event lemmas (denoted as lemmas{t}) are produced. Forward and backward subsumption are included.

Program 1 Community revision process.

```
community-revision (abb{t-1}, event{t}, MAX_ADMISSIBLE_DEPTH)
{
 temp_ABB = abb{t-1};
 SOS = move_to_SOS(convert_to_CNF(event{t}));
 temp_ABB = move_to_ABB(SOS);
 while (SOS!=NULL) do
 {
   actual_clause = choose_the_first(SOS);
   SOS = remove_from_SOS(actual_clause);
   new_clauses = generate_new_clauses(actual_clause, temp_ABB);
   for (actual_new_clause in new_clauses) do
   {
     if ((actual_new_clause != NULL)
         and not_tautology(actual_new_clause)
         and not_subsumed(temp_ABB, actual_new_clause)
         and correct_with_parents(actual_new_clause)
        )
     if (depth(actual_new_clause) < MAX_ADMISSIBLE_DEPTH)
       SOS = move_to_SOS(actual_new_clause);
     temp_ABB = move_to_ABB(actual_new_clause);
     if unit_clause?(actual_new_clause)
       temp_lemmas = move_to_lemmas(actual_new_clause);
     for clause_in_ABB in temp_ABB do
     {
       if subsumes(actual_new_clause, clause_in_ABB)
       {
         temp_ABB = remove(clause_in_ABB, temp_ABB);
         if (temp_ABB in SOS)
          SOS = remove(clause_in_ABB, SOS);
       }
     }
   }
 }
 abb{t} = temp_ABB;
 lemmas{t} = temp_lemmas;
}
```

The function `move_to_SOS` appends either a clause or a set of clauses to the `SOS` list. Other list functions append either a clause or a set of clauses to the specific list only. The function `generate_new_clauses` generates new clauses using resolution over all literals in actual clause and over all clauses (literals) in the `temp_ABB` list. We process the new clause only if:

- the new clause is not empty clause (we do not suppose no-monotonicity in the **revise***time*, see $\mathsf{model}_0^S(\theta)$ constitution, where can appear inconsistent events within searching for all reachable lemmas),

- the new clause is not a tautology,
- the new clause is not subsumed by another one in `temp_ABB` – forward subsumption process and
- the length of the new clauses is lower than any parent clause, the length of the clause is here understood as a number of the symbols in the clause.

There is backward subsumption process where we try whether the clauses in `temp_ABB` are subsumed by the new clause.

We proposed in [2] that the community revision process should be close to $\uplus^{\min}$ than to $\uplus^{\max}$. The reason is an exponential complexity of amount of new deduced knowledge when the event belief formula appears. We have experimented with admissible depth of the clause derivation within the **revise***time*.

4.2.2 Automated Reasoning in Inspection Time

The community inspection operator $\looparrowright$ is implemented in three steps as follows:

Example 5 Community revision process.

Let us continue in example 4. Coming sequential event belief formulae are:

$\mathsf{ebb}_3(\theta) =$
$\quad \mathtt{event}_1 = \neg accept(\text{"}Country - C\ Army\text{"},\text{"}Country - C\ Police\text{"},$
$\quad\quad \text{"}Suffer\ Town\text{"}),$
$\quad \mathtt{event}_2 = accept(\text{"}Country - C\ Army\text{"},\text{"}Country - A\ Army\text{"},$
$\quad\quad \text{"}Suffer\ Town\text{"}),$
$\quad \mathtt{event}_3 = proposal(\text{"}Country - C\ Police\text{"},\text{"}Country - C\ Army\text{"})\}.$

Let us simulate the community revision process and let us observe contents of the sets $\mathsf{abb}_t(\theta)$ and $\mathsf{event\_lemmas}_t(\theta)$:

$t = 0$
$\quad \mathsf{abb}_0(\theta) = \mathsf{model}_0^S(\theta),$
$\quad \mathsf{event\_lemmas}_0(\theta) = \emptyset,$
$t = 1$
$\quad \mathtt{event}_1 =$
$\quad\quad \neg accept(\text{"}Country - C\ Army\text{"},\text{"}Country - C\ Police\text{"},\text{"}Suffer\ Town\text{"})$
$\quad \mathsf{abb}_1(\theta) = \mathsf{model}_0^S(\theta) \cup$
$\quad\quad \{\neg accept\_leader(\text{"}Country - C\ Army\text{"},\text{"}Country - C\ Police\text{"})$
$\quad\quad\quad \vee \neg accept\_city(\text{"}Country - C\ Army\text{"},\text{"}Suffer\ Town\text{"})$
$\quad\quad \}$
$\quad \mathsf{event\_lemmas}_1(\theta) = \emptyset,$
$t = 2$
$\quad \mathtt{event}_2 =$
$\quad\quad accept(\text{"}Country - C\ Army\text{"},\text{"}Country - A\ Army\text{"},\text{"}Suffer\ Town\text{"})$
$\quad \mathsf{abb}_2(\theta) = \mathsf{model}_0^S(\theta) \cup$
$\quad\quad \{accept\_leader(\text{"}Country - C\ Army\text{"},\text{"}Country - A\ Army\text{"}),$
$\quad\quad\quad accept\_city(\text{"}Country - C\ Army\text{"},\text{"}Suffer\ Town\text{"}),$,
$\quad\quad\quad \neg accept\_leader(\text{"}Country - C\ Army\text{"},\text{"}Country - C\ Police\text{"})$
$\quad\quad \}$
$\quad$ clause:
$\quad\quad \neg accept\_leader(\text{"}Country - C\ Army\text{"},\text{"}Country - C\ Police\text{"})$
$\quad\quad \vee \neg accept\_city(\text{"}Country - C\ Army\text{"},\text{"}Suffer\ Town\text{"})$
$\quad$ has been removed by backward subsumption process,
$\quad \mathsf{event\_lemmas}_2(\theta) =$
$\quad\quad \{$
$\quad\quad\quad accept\_leader(\text{"}Country - C\ Army\text{"},\text{"}Country - A\ Army\text{"}),$
$\quad\quad\quad accept\_city(\text{"}Country - C\ Army\text{"},\text{"}Suffer\ Town\text{"}),$
$\quad\quad\quad \neg accept\_leader(\text{"}Country - C\ Army\text{"},\text{"}Country - C\ Police\text{"})$
$\quad\quad \}$
$t = 3$
$\quad \mathtt{event}_3 = proposal(\text{"}Country - C\ Police\text{"},\text{"}Country - C\ Army\text{"})$, without the impact to the observed sets:
$\quad \mathsf{abb}_3(\theta) = \mathsf{abb}_2(\theta),$
$\quad \mathsf{event\_lemmas}_3(\theta) = \mathsf{event\_lemmas}_2(\theta).$

1. if $\mathtt{goal}_t \in \mathsf{model}_t(\theta)$ the reply is *yes* and finish, otherwise continue in step 2,
2. if $\mathsf{model}_t(\theta) \vdash \mathtt{goal}_t$ the reply is *yes* and finish, otherwise continue in step 3,
3. the reply is *no*.

We will focus on the inspection time supported by $\uplus^{\min}$ community revision operator and supported by $\uplus^{N}$ community revision operator. Both require theorem proving methods in the **inspect***time*.

4.2.3 Community Model Inspection

We used the resolution principle, exactly binary resolution for our experiments with theorem proving and tools related to it. We assume that $\mathsf{model}_t(\theta) = \mathsf{ebb}_t(\theta) \cup \mathsf{gbb}$. If the $\mathtt{goal}_t$ formula is not directly contained in the $\mathsf{model}_t(\theta)$, then the problem becomes that of provability: $\mathsf{model}_t(\theta) \vdash \mathtt{goal}_t$. The actual model $\mathsf{model}_t(\theta)$ is kept in CNF form. The user's query `goal` is converted to the CNF form too. The $\nsim$ operator is implemented by finding inconsistencies in $\mathsf{model}_t(\theta) \cup \neg\mathtt{goal}_t$.

We studied different searching algorithms and automated reasoning techniques for implementation of the community inspection process in our domain. We experimented with a combination of several searching strategies too. Lemmatizing, caching and deletion with lemmas techniques are used.

Searching Strategy

Our experiments confirm that the set of support strategy works best for our problem [25]. Theorem prover based on set of support strategy can be found in OTTER theorem prover [21]. Forward and backward subsumption can be included. As most of the searching algorithms, we store following lists within the searching algorithm:

- `usable` – this list contains clauses that are available to make inferences,
- `sos` – clauses in list sos (set of support) are not available to make inferences, they are waiting to participate in the search via expansion process due to the resolution principle, it is equivalent to `OPEN` list used in problem solving searching algorithms,
- `close` – clauses prepared for forward subsumption process, they could subsume a new generated clause, it is equivalent to `CLOSE` list used in problem solving searching algorithms.

See program 2 for basic searching algorithm description.

The searching strategy is similar to A^* algorithm [23]. The function `move_to_USABLE` appends either a clause or a set of clauses to the `USABLE` list and it makes available every literal in the clause for inference. Other list functions append either a clause or a set of clauses to the specific list only. The

Program 2 Searching algorithm.

```
search1 (query, theory)
{
 USABLE = move_to_USABLE(theory);
 SOS = move_to_SOS(convert_to_CNF(not query));
 CLOSE = append(USABLE, SOS);
 evaluate_clauses(SOS);
 while (SOS!=NULL) do
 {
   actual_clause = choose_the_cheapest(SOS);
   SOS = remove_from_SOS(actual_clause);
   if (actual_clause == NIL) return T;
   new_clauses = generate_new_clauses(actual_clause, USABLE);
   for (actual_new_clause in new_clauses) do
   {
     if (not_tautology(actual_new_clause)
         and not_subsumed(CLOSE, actual_new_clause)
        )
     evaluate_clause(clause);
     SOS = move_to_SOS(actual_new_clause);
     CLOSE = move_to_CLOSE(actual_new_clause);
   }
   USABLE = move_to_USABLE(actual_clause);
 }
}
```

functions `evaluate_clauses` and `evaluate_clause` assign to clauses, respectively to clause the objective function, which estimates number of operations until empty clause will be reached. We prefer easy estimation – the number of literals. The function `generate_new_clauses` generates new clauses using resolution over all literals in actual clause and over all clauses (literals) in the `USABLE` list. The `CLOSE` list enables forward subsumption process. Backward subsumption process is not included in the program description.

Model Elimination

We applied model elimination features within the set of support searching strategy. Expansion process via resolution principle is performed over only one literal – the rightmost literal in the expanded clause. We will try to eliminate this the rightmost literal – it becomes the newest A-literal. The rest of the second resolvent (not expanded clause) is appended to the rest of expanded clause from the right – there are new B-literals. The new clause inherits from the parent expanded clause list of their A-literals (literals with their positions). Later, when the new clause is expanded, there is set up the newest A-literal. If there is no B-literal behind any A-literal, the A-literal

is eliminated and removed. See example 6 for several resolution steps. See program 3 for the new searching algorithm description:

Program 3 Searching algorithm.

```
search2 (query, theory)
{
 USABLE = move_to_USABLE(theory);
 SOS = move_to_SOS(convert_to_CNF(not query));
 CLOSE = append(USABLE, SOS);
 evaluate_clauses(SOS);
 while (SOS!=NULL) do
 {
   actual_clause = choose_the_cheapest(SOS);
   SOS = remove_from_SOS(actual_clause);
   if (actual_clause == NIL) return T;
   CLOSE = move_to_CLOSE(actual_clause);
   new_clauses = generate_new_clauses_ME(actual_clause, USABLE);
   for (actual_new_clause in new_clauses) do
   {
     if (not_tautology(actual_new_clause)
         and not_subsumed(CLOSE, actual_new_clause)
        )
     evaluate_clause(clause);
     SOS = move_to_SOS(actual_new_clause);
     CLOSE = move_to_CLOSE(actual_new_clause);
   }
   USABLE = move_to_USABLE_ME(actual_clause);
 }
}
```

The function `generate_new_clauses` generates new clauses using resolution over the rightmost literal in actual clause and over all clauses in the `USABLE` list. Be aware of the fact that the clauses in the `USABLE` list, which are supported by the query they have only one literal available for the inferences – it is the newest A-literal. The function `move_to_USABLE_ME` appends either a clause or a set of clauses to the `USABLE` list and it makes available the rightmost literal in clause for inference only.

We have designed and implemented above described approach because when the clause is expanded over all its literals then too many new clauses are generated. The new clauses differ in two literals (due to the binary resolution) and unifications only. We suppose that by the model elimination support we will improve inspection phase due to the elimination of "similar" clauses generation.

We were inspired by [30] and we improved above the described strategy in such way that the newest A-literal need not to be the rightmost literal (let

us denote it as literal preference). The newest A-literal is chosen by heuristic knowledge. We experimented with several evaluation functions for choosing the newest A-literal. With respect to reduction of the searched space by caching (see below) and by reduction of the number of resolution candidates we prefer this heuristic: the ratio between the amount of variables and the amount of constants – we prefer the literal with the lowest ratio. The chance that the newest A-literal will be subsumed by more general lemma increases with decreasing ratio between the amount of variables and the amount of constants. The newest A-literal is chosen from all B-literals behind either the last A-literal or from all literals if no A-literal presents in the clause. The newest A-literal is removed from the clause and then appended to the end of the clause. See an example 6:

See program 4 for the improved searching algorithm.

The function `set_the_newest_A_literal` sets up the newest A-literal and transforms actual clause so that the newest A-literal is the last literal in the clause.

We suppose that by choosing the newest A-literal from all B-literals behind the last A-literal:

- we will improve the efficiency of the inspection phase due to the ability to control searching algorithm via the possibility to choose the new eliminated literal and
- we will improve the efficiency of the inspection phase due to the more frequent using of caching (see below).

Proof Lemmas

Besides the $\mathsf{event\_lemmas}_t(\theta)$ that are deduced unit clauses formulated by the $\uplus$ operators in the **revise***time*, let us introduce $\mathsf{proof\_lemmas}_t(\theta)$ as a set of unit clauses (below proof lemmas) generated by the $\looparrowright$ during the **inspect***time* (theorem proving process) as a side effect. We are able to generate proof lemmas within the proofs due to the fact that we have combined searching strategy with model elimination method. See [31] for proof lemmas generation when only model elimination method is used.

Let us suppose now that the newest A-literal in some clause is eliminated. We could execute a new community inspection process:

$$\mathsf{model}_t(\theta) \vdash \mathsf{A-literal}. \tag{4.3}$$

We can say that:

$$\mathsf{model}_t(\theta) \models \mathsf{A-literal}, \tag{4.4}$$

and we can eliminate the newest A-literal if the new community inspection process succeeds. We cannot apply this approach with respect to two reasons:

- the community inspection process suggests only one solution if it succeeds then the original community inspection process, which invokes next one in order to eliminate the newest A-literal, would not be complete and

Example 6 Model elimination.

1. let us expand the clause:
   $accept(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},Z)\vee$
   $\neg accept\_leader(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"})$
   with clause:
   $accept\_leader(X,Y)\vee proposal(Y,X)\vee refuse\_proposal(X,Y,M)$,
   there is no A-literal and two B-literals in clause:
   $accept(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},Z)\vee$
   $\neg accept\_leader(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"})$,
   the newest A-literal is set up:
   $\neg accept\_leader(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"})$
   because the ratio between the amount of variables and amount of constants is lower than in the case of literal:
   $accept(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},Z)$,
2. a new clause is obtained:
   $accept(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},Z)$
   $[\vee\neg accept\_leader(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"})]$
   $\vee proposal(\text{"}Country-C\ Police\text{"},\text{"}Country-C\ Army\text{"})$
   $\vee refuse\_proposal(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},M)$,
   one A-literal has been inherited from the parent clause (bracketed literal), there are three B-literals, we choose the newest A-literal between two B-literals behind the last A-literal, the newest A-literal is:
   $proposal(\text{"}Country-C\ Police\text{"},\text{"}Country-C\ Army\text{"})$,
   the clause is repaired to the form:
   $accept(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},Z)$
   $[\vee\neg accept\_leader(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"})]$
   $\vee refuse\_proposal(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},M)$
   $\vee proposal(\text{"}Country-C\ Police\text{"},\text{"}Country-C\ Army\text{"})$,
3. let us expand the clause:
   $accept(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},Z)$
   $[\vee\neg accept\_leader(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"})]$
   $\vee refuse\_proposal(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},M)$
   $\vee proposal(\text{"}Country-C\ Police\text{"},\text{"}Country-C\ Army\text{"})$
   with clause:
   $\neg proposal(\text{"}Country-C\ Police\text{"},\text{"}Country-C\ Army\text{"})$,
4. a new clause is obtained:
   $accept(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},Z)$
   $[\vee\neg accept\_leader(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"})]$
   $\vee refuse\_proposal(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},M)$,
   $[\vee proposal(\text{"}Country-C\ Police\text{"},\text{"}Country-C\ Army\text{"})]$,
   there are two A-literals and two B-literals, there is no B-literal behind the last A-literal, then it is eliminated, the clause is transformed to the form:
   $accept(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},Z)$
   $[\vee\neg accept\_leader(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"})]$
   $\vee refuse\_proposal(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},M)$,
   the newest A-literal is (there is no choice):
   $refuse\_proposal(\text{"}Country-C\ Army\text{"},\text{"}Country-C\ Police\text{"},M)$,

Program 4 Searching algorithm.

```
search3 (query, theory)
{
 USABLE = move_to_USABLE(theory);
 SOS = move_to_SOS(convert_to_CNF(not query));
 CLOSE = append(USABLE, SOS);
 evaluate_clauses(SOS);
 while (SOS!=NULL) do
 {
   actual_clause = choose_the_cheapest(SOS);
   SOS = remove_from_SOS(actual_clause);
   if (actual_clause == NIL) return T;
   CLOSE = move_to_CLOSE(actual_clause);
   set_the_newest_A_literal(actual_clause);
   new_clauses = generate_new_clauses_ME(actual_clause, USABLE);
   for (actual_new_clause in new_clauses) do
   {
     if (not_tautology(actual_new_clause)
         and not_subsumed(CLOSE, actual_new_clause)
        )
     evaluate_clause(clause);
     SOS = move_to_SOS(actual_new_clause);
     CLOSE = move_to_CLOSE(actual_new_clause);
   }
   USABLE = move_to_USABLE_ME(actual_clause);
 }
}
```

- whole the community inspection process would not be such efficient with respect to the redundancy of derived clauses due to less efficient subsumption process.

However, we generate proof lemmas under conditions resulting from the requirements for the isolation of the community revision processes in above described approach. The proof lemma is generated when any A-literal is eliminated and if:

1. only one parent of the clauses generated between the clause where the A-literal has been set and the clause where the A-literal is eliminated, is supported by goal hypothesis and
2. no B-literal ahead of eliminated A-literal has influenced B-literals behind eliminated A-literal (some B-literals can be removed from the new clause because they appear several times in the clause).

A new proof lemma as a new deduced knowledge is true fact in $\mathsf{model}_t(\theta)$ (see eqn.4.4). Proof lemmas can be used either:

- within the same proof where they are generated through technologies as lemmatizing, caching and deletion with proof lemmas or
- within following proofs.

Lemmatizing and Caching

We were inspired by [32, 31] where iterative deepening search engine has been supported by caching and lemmatizing. When caching we replace the parts of the searched state space, when lemmatizing we suggest to the searching algorithms sub-solutions. We prefer caching to lemmatizing with respect to knowledge redundancy if lemmatizing is used. We experimented with both improvement technologies.

Let us suppose now that a lemma is used as a parent to generate a new clause. Is it caching or lemmatizing? We are able to say whether the lemma will be used within lemmatizing or chacing according to the expansion over A-literal. If the lemma subsumes (is more general) than expanded A-literal then caching is applied otherwise lemmatizing is applied.

We do not consider other candidates if lemma is applied as caching, because a new clauses is shorter then expanded parent clause and the new clause subsumes expanded parent clause (no binding has been performed). Lemmatizing process [32] brings the knowledge redundancy to the proof which can result even in deceleration of the theorem proving process. In the worst case, some clauses are generated two times, but they are reduced immediately by the subsumption process. These clauses are produced by lemmatizing process. Our domain, as most of meta-reasoning domains, is characterized by low number of variables. Subsumption process removes the redundancy in the search space very early then we prefer the lemmas to other candidates for expansion process via resolution principle. The process generating the candidates for expansion process is finished when lemma supporting the caching is found. We prefer the lemmas supporting lemmatizing to other candidates except these which supports caching.

We suppose that caching and lemmatizing improve the community inspection phase and we experimented with it. It does need not to be a true because it is necessary to consider the fact that there are additional operations for the proof lemmas generation and the fact of knowledge redundancy. In cases when the proof lemmas will not be such efficient, the community inspection process can be slow down.

Deletion with Lemmas

We suggest a novel utilization of the lemmas based on the deletion strategy. We reduce the space of all clauses by removing these which are subsumed by any proof lemma. Proof lemmas for deletion of the clauses can be used in two ways:

- in backward subsumption process, we try to remove all clauses from `USABLE` and `SOS` lists which are subsumed by a new found proof lemma,
- in forward subsumption process, when a new found proof lemmas is put into the list `CLOSE`.

We suppose that deletion supported by proof lemmas will reduce the space of all clauses and then time responses of the community inspection phase.

4.2.4 Community Model Inspection Supported by $\uplus^{N}$.

We can apply here all techniques described above. Additionally, we generate lemmas within the **revise***time*. Let us denote $\mathsf{lemmas}_t(\theta)$ as set of all lemmas deduced within the **revise***time* and within the proofs:

$$\mathsf{lemmas}_t(\theta) = \mathsf{event\_lemmas}_t(\theta) \cup \mathsf{proof\_lemmas}_t(\theta). \tag{4.5}$$

The community model inspection is implemented as a proof as follows:

$$\mathsf{model}_t(\theta) \cup \mathsf{lemmas}_t(\theta) \vdash \mathtt{goal}_t. \tag{4.6}$$

We will experiment with the lemmas deduced within the **revise***time* and with relation between $\uplus$ and $\looparrowright$ operators. We will focus on the amount of deduced lemmas and their utilization. We suppose that lemmas generated within the proof are more useful than lemmas deduced within the **revise***time*, because they are generated as a result of the user's query. The chance that proof lemma will be successfully applied as either caching or lemmatizing is higher than in the case of event lemma. It is possible that the set $\mathsf{lemmas}_t(\theta)$ will be so big that using of lemmas will have negative impact to the community inspection phase. The methodologies how to reduce the $\mathsf{lemmas}_t(\theta)$ set by identification of useful lemmas is in the scope of further research.

Our task differs from the classical theorem proving task by the fact, that we are able to divide the clauses submitted to the theorem proving process into three disjunctive parts: gbb, $\mathsf{ebb}_t(\theta)$ and $\mathsf{lemmas}_t(\theta)$. We found that the different settings of the `CLOSE` list used in searching algorithm have different results. That is why we will experiment with different settings of the searching algorithm in order to find out the fastest prototype for our domain.

4.3 Induction

One of the biggest problems in implementation of the explicit models is the representation of all created facts with a possibility to search trough them quickly, because the space of all facts logically following from the events could be large and even infinite. One possible solution is to work only with the implicit representation – representation of hypotheses about the object agent's decision making algorithm. Therefore, we are faced with an inverse task to the

agent's decision making process described above (section 1.5): *Assuming the knowledge of* gbb*, identify ψ_A by observing agent A's decisions.* From now, we will work with the model containing information only about one object agent, the model of the whole object-level community can be obtained by composing these models.

4.3.1 Version Space

During the community model revision phase (**revise***time*), we can create hypotheses ψ_A^* about the agent's A decision making algorithm ψ_A. A good way to manipulate the space of hypotheses is provided by a version space algorithm (VS) [33], where all consistent hypotheses are represented by two sets containing the most general and the most specific hypotheses.

Our hypothesis ψ_A^* about ψ_A will be composed of n elementary hypotheses ${\psi_A^*}^i$ each deciding about one elementary attribute τ^i of the task τ, $i = 1, \ldots, n$:

$$\psi_A^*(\tau) = \left[{\psi_A^*}^i(\tau^i) \right]_{i=1}^n , \tag{4.7}$$

and it will be used as follows:

$$\psi_A^*(\tau) \rightarrow \begin{cases} yes & \text{if } \bigwedge_i {\psi_A^*}^i(\tau^i) = yes \\ no & \text{otherwise.} \end{cases} \tag{4.8}$$

This equation is equivalent to:

$$\psi_A^*(\tau) \rightarrow \begin{cases} yes & \text{if } \bigvee_i \overline{\psi^{*i}}(\tau^i) = no, \\ no & \text{otherwise.} \end{cases} \tag{4.9}$$

where $\overline{\psi^{*i}}$ is complementary hypothesis[1] to ψ^{*i}. Therefore, usage of complementary hypotheses enables VS to learn more complex hypothesis about object agent's decisions, even if it works only with conjunction of elementary hypothesis.

Version Space Algorithm Description.

Algorithm VS represents the space of all hypotheses by two sets:

- $\mathcal{G}$ – the set of the most general consistent hypotheses and
- $\mathcal{S}$ – the set of the most specific consistent hypotheses,

so that all and only those hypotheses, that are more general than some hypothesis in $\mathcal{S}$ and more specific than some hypothesis in $\mathcal{G}$ are consistent with processed events. Therefore the model managed by VS algorithm is a pair:

[1] A hypothesis ψ_1^* is complementary to the hypothesis ψ_2^* if it replies *yes* when ψ_2^* replies *no* and vice versa.

$$\mathsf{model}^{\mathsf{VS}} = [\mathcal{S}, \mathcal{G}] \tag{4.10}$$

The meta-agent uses the VS algorithm in the following manners:

init*time*: The initialization of VS algorithm is done by (i) initializing the set $\mathcal{G}$ to contain the most general hypothesis – always responding *yes* and (ii) initializing the set $\mathcal{S}$ to contain a hypothesis that always responds *no*.

revise*time*: Incoming event (represented by a pair $[\tau_t, \psi(\tau_t)]$ as described in section 2.4, eqn. 2.12) is processed in as follows:

$$\mathsf{model}_t^{\mathsf{VS}} \uplus \mathtt{event}_t \rightarrow \begin{cases} \mathtt{Generalize}\ \mathcal{S}, \mathtt{Remove}\ \mathcal{G} \\ \quad \text{if } \psi(\tau_t) = yes, \\ \mathtt{Specify}\ \mathcal{G}, \mathtt{Remove}\ \mathcal{S} \\ \quad \text{if } \psi(\tau_t) = no, \end{cases} \tag{4.11}$$

where the operation $\mathtt{Generalize}\ \mathcal{S}$ ($\mathtt{Specify}\ \mathcal{G}$) means to generalize (specify) those hypotheses in the set $\mathcal{S}$ ($\mathcal{G}$) that not comply with the given event `event` and the operation `Remove` removes wrong hypotheses from the given set.

The community revision phase can be stopped in two cases:

- If one of the sets $\mathcal{G}$ and $\mathcal{S}$ is empty, we end because the given events were inconsistent.
- If the sets $\mathcal{G}$ and $\mathcal{S}$ are singleton and identical, we end because we have found the exact decision rule of the object agent.

inspect*time*: The algorithm is predicting the response for a given query $\mathtt{goal}_t$ as follows:

$$\mathsf{model}_t^{\mathsf{VS}} \looparrowright \mathtt{goal}_t \rightarrow \begin{cases} yes & \text{if all hypotheses in } \mathcal{G} \text{ and } \mathcal{S} \\ & \text{evaluate the } \mathtt{goal_t} \text{ to be positive,} \\ no & \text{if all hypotheses in } \mathcal{G} \text{ and } \mathcal{S} \\ & \text{evaluate the } \mathtt{goal_t} \text{ to be negative,} \\ unsure & \text{otherwise.} \end{cases} \tag{4.12}$$

Studying positive and negative evaluations of the hypotheses allows also to count the probabilities of unsure queries.

4.3.2 Inductive Logic Programming

This section deals with meta-reasoning based on *Inductive Logic Programming (ILP)* [34]. First, we briefly explain the ILP background and the first-order logic representation of agent's decision making. We then describe the basic variant of ILP meta-reasoning operations and conclude with a few remarks regarding ILP meta-reasoning reliability, efficiency and its possible alternative implementation.

The task of creating hypotheses about agent's decision rule can be accomplished by means of an ILP system, whose input is actual object agent's model $\mathsf{model}_t^{\mathsf{ILP}}$ in the time t:

$$\mathsf{model}_t^{\mathsf{ILP}} = \mathsf{gbb} \cup \mathsf{ebb}_\mathsf{t} \tag{4.13}$$

The event belief base ebb_t contains history of A's decisions up to the instant t. Each item in the history is a pair $[\tau_t, \psi(\tau_t)]$ (described in section 2.4, eqn. 2.12).

The output of the ILP algorithm is then a hypothesis ψ_A^*, which approximates the unknown correct ψ_A. Once, we have the hypothesis ψ_A^*, we can use it to predict A's decision during model inspection. The hypothesis is expressed in the programming language Prolog.

ILP Meta-Reasoning Operations

This section describes the basic variant of ILP meta-reasoning operations with respect to the three meta-reasoning phases. For discussion and the description of alternative approaches, see the end of this section.

ILP algorithm is applied to create a hypothesis. In ILP terminology, it is called **hypothesis induction**. Given A's decision history (stored in ebb) and background belief base gbb, an ILP system is employed to induce a hypothesis ψ_A^* about ψ_A the A's decision making algorithm. This is ensured by an ILP system Aleph[2][35]. This induction can be carried out in the **revise***time*, in the **inspect***time* or in the idle time after new event comes.

init*time*

Agent A's decision history is initialized by setting:

$$\mathsf{model}_0^{\mathsf{ILP}} = \mathsf{gbb} \tag{4.14}$$

revise*time*

In its basic form, the ILP $\uplus$ operator consists solely of updating A's decision history with a newly observed decision, i.e.:

$$\mathsf{model}_t^{\mathsf{ILP}} \uplus \mathtt{event}_t = \mathsf{model}_t^{\mathsf{ILP}} \cup \{[\tau_t, \psi(\tau_t)]\} \tag{4.15}$$

This revision creates the model as defined in equation 4.13. The ILP $\uplus$ operator is equivalent to the community revision operator $\uplus^{\min}$ defined in eqn. 2.7.

inspect*time*

Before the $\mathsf{model}_t^{\mathsf{ILP}}$ can reply given query, it is necessary to induce a hypothesis ψ_A^*:

$$\psi_A^* = Aleph\left(\mathsf{model}_t^{\mathsf{ILP}}\right). \tag{4.16}$$

The induced hypothesis ψ_A^* is used to answer the query $\mathtt{goal}_t$ by performing a resolution operation. Specifically,

[2] `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph`

$$\mathsf{model}_t^{\mathsf{ILP}} \looparrowright \mathtt{goal}_t \rightarrow \begin{cases} \textit{yes} & \text{if } \psi_A^* \vdash \textit{accept}(\mathtt{goal}_t), \\ \textit{no} & \text{if the attempt to prove } \textit{accept}(\mathtt{goal}_t) \\ & \text{ends by finite failure,} \\ \textit{unsure} & \text{otherwise.} \end{cases} \tag{4.17}$$

Important Remarks

Reliability of ILP Predictions

Regarding predictions obtained via the meta-reasoning process, there is one significant difference between ILP and Version Space (VS) 4.3.1 algorithm. Any ILP algorithm searches for complete and consistent generalization of the seen examples. This means that the ILP meta-reasoning can return wrong predictions for goal queries which have not been so far observed (and which thus have not been used in the **revise***time*).

Possible incorrect predictions on previously unseen event belief formulae[3] is an inherent property of all systems performing process of the generalization of past event belief formulae.

Reusability of Induced Hypotheses

The induced hypothesis *accept* can be reused for all subsequent queries concerning A's decision making as long as there have not been any new event belief formulae. If new event belief formula is inconsistent with the existing ψ_A^* hypothesis, the hypothesis induction operation has to be re-executed, i.e. the completely new hypothesis has to be generated from the $\mathsf{gbb} \cup \mathsf{ebb_t}$.

*Implementation of ILP Meta-Reasoning **revise**time versus **inspect**time Operations*

The above given remark leads to an important question regarding how the core ILP meta-reasoning operations, i.e., hypothesis induction and query resolution should be distributed between the **revise***time* and the **inspect***time*. ILP algorithms are considered to be complex and since demanding. That is why ILP systems have often problems with the respond time, which seems too long for specific applications. Recently, there were suggested some heuristics [35] which can improve this situation. In the current implementation, both operations are executed in the **inspect***time* and the **revise***time* operation solely updates object agent's decision history.

Our experiments in the considered domain proved that here call of ILP algorithm causes no time-delay. Consequently, an alternative solution is possible. The hypothesis induction operation could be moved to the **revise***time*,

[3] In contrast to the predictions on testing examples, the hypothesis induced by an ILP algorithm is always consistent with the training set, unless the training set is itself inconsistent.

i.e., the ψ^*_A hypothesis would be automatically re-induced each time a new event belief formula about the object agent's decision is observed. In this case, only the query resolution operation is left for the **inspect***time*. Since the resolution is generally much faster than the induction, such solution would result in a faster model inspection. On the other hand, the overall amount of computation carried out when adopting this solution can be (substantially) higher than using the original one. This is because the hypothesis inductionstep would be performed even if not utilized by a pending inspection query.

Batch versus Incremental Induction

The hypothesis induction operation can be realized either by a batch or by an incremental ILP system. Batch ILP systems induce the theory only after they have all training set at their disposal. Incremental ILP systems, also called *theory revision* systems [36], induce the theory in an example-by-example manner, each time updating the so far induced theory so that it reflects the newly processed event belief formula. In a broader perspective, the batch and incremental induction corresponds to the *strong-update* and *weak-update* strategies, respectively.

Weak update strategy is generally preferred as it is supposed to require less computing. In the case of ILP, the situation is reversed. Since the mid 1990s, the interest of the ILP research community has moved away from theory-revision systems.[4] We used batch ILP systems for our ILP meta-reasoning implementation.

[4] Which also means that there is lack of reliable first-order theory revision systems, which we could employ in our implementation.

5

Experiments

We performed a large number of experiments in order to verify used technologies and algorithms. The most interesting graphs are presented in the following section.

First we will compare how successfully the three reasoning technologies – theorem proving, version space machine learning and inductive logic programming – can predict community behavior 5.2. We have primarily paid attention to deductive reasoning, therefore the major part of the experiments will be devoted to describing properties of the suggested theorem proving mechanism that we have implemented.

We will compare behavior of different community revision operators in deductive meta-reasoning. We will try to illustrate how much computation is it useful and rational (in terms of calculative rationality) to carry out in the community revision and inspection phase respectively (in other words, how close to $\uplus^{\max}$ is it sensible to get).

In the following section we will illustrate the role of the domain knowledge in the inspection time.

After that we will present a series of four measurement that will illustrate the behavior of the reasoning improvements that we have designed and explained in sections 4.2.3.

The experiments will conclude with the comparison of our deduction meta-reasoning methods and the known and cited Otter theorem prover.

5.1 Description of Experiments

Let us describe the language used by the automated reasoning part of the meta-agent. We use the first order logic without equality and function symbols. We utilize the relation operator $<$. The ground belief base `gbb` consists of 20 formulae which are transformed to 249 clauses. See *automated-reasoning-gbb.txt* file for `gbb`.

See *theorem-proving-queries.txt* file for definition of 21 queries used for most reported experiments. The only exception is the experiment in section 5.2, for experiment description see 5.2.

We defined the set of the disasters sent to the multi-agent system CPlanT in order to obtain the event belief formulae for our experiments. See *theorem-proving-events.txt* file for the event belief formulae.

All experiments have been performed with events stored in *theorem-proving-events.txt*. Only experiment with simulated reasoning and abilities of the meta-reasoning methods to predict the object agent's decision making has been performed with event belief formulae stored in *prediction-events.txt*.

5.2 Prediction Capabilities of Investigated Methods

We experimented with an ability to predict the object agent's decision making. We defined a specific set of the disasters sent to the CPlanT multi-agent system and we tried to predict if the object agent would either accept or refuse the team allocation request. First we try to predict the event if the event is equivalent to the the team allocation request. Then the event is suggested to the meta-reasoning methods.

See graph 5.1 for the success of the prediction with respect to the meta-reasoning methods:

- automated reasoning and theorem proving (see 4.2),
- automated reasoning and theorem proving supported by event belief formulae generated by the reasoning simulation (see 4.1 and 4.2),
- machine learning, version space algorithm (see 4.3.1),
- machine learning, inductive logic programming (see 4.3.2).

The machine learning methods have not been supported by event belief formulae generated by the reasoning simulation. To evaluate the results we have performed revision and inspection of 160 sequential events. Percentage of correct predictions is counted during last 20 events. See graph 5.2 for the comparison of the meta-reasoning methods in prediction.

Reasoning simulation improves the ability to predict the object agent's decision making due to the generation of more knowledge about the object agents. The inductive logic programming method seems to reach the best results due to the ability to generalize knowledge via the induction operation. On the other hand there ILP gave five incorrect predictions. There are no incorrect predictions if we use the automated reasoning method based on the deduction operation. The version space method does not generate incorrect predictions although the induction operation is used here, because it stores all possible hypothesis consistent with the previous events. The results of the automated reasoning (without the support of the reasoning simulation) and version space methods are similar. The version space method uses the same background knowledge as the automated reasoning method. While the

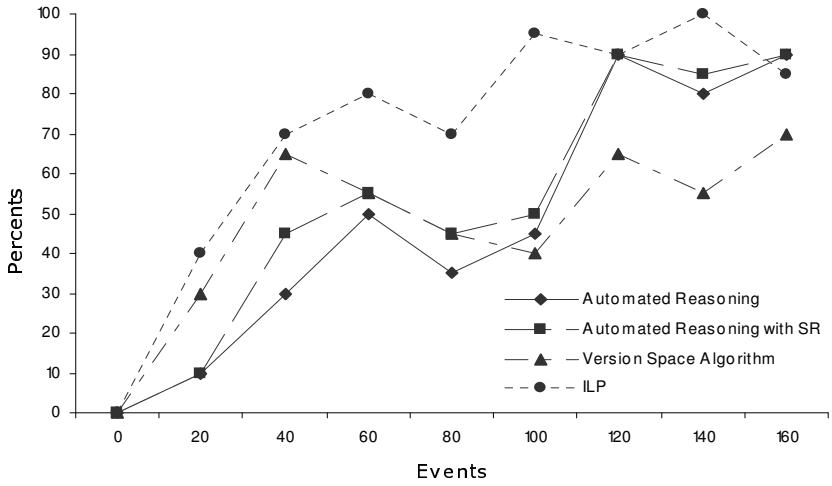

Fig. 5.1. Success of Prediction.

automated reasoning method uses background knowledge in explicit form, version space method uses background knowledge in implicit form.

The success of the prediction can decrease although in global measurement it increases. The prediction success of specific event depends on previous event, namely on its similarity to the past events. With an amount of new deduced knowledge from the event it decreases the probability that we will be able to predict the event.

In future we intend to combine the methods. The methods based on deduction and version space algorithm give correct predictions. We utilize inductive methods if no deductive method gives the prediction. We will risk incorrect prediction in some cases to no prediction.

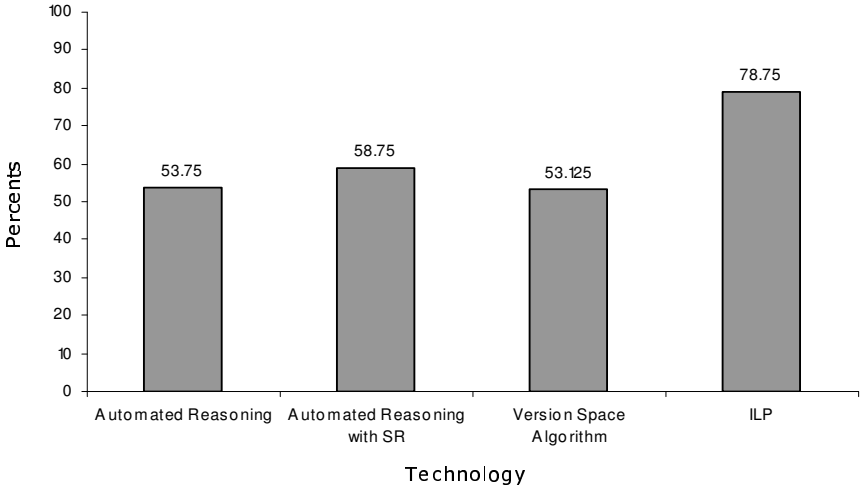

Fig. 5.2. Comparison of Meta-reasoning Methods in Prediction.

5.3 Community Revision Operators

We studied the impact of the community revision operator to the **inspect***time*. We experimented with different community revision operators defined via the maximal admissible depth of the clause derivation within the **revise***time* operation (see section 4.2.1). See graph 5.3 for the number of the generated clauses within the **inspect***time*. We accumulated 21 results of the proofs supported by the event lemmas generated within the **revise***time*. The proofs have been supported by the technologies *caching*, *lemmatizing* and *deletion with lemmas*. There are the community revision operators (there is 249 clauses in gbb):

- CRO0 - $\uplus^{\min}$ community revision operator,
- CRO2 - $\uplus^{7}$ community revision operator, the maximal admissible depth of the clause derivation equals to 2, (see section 4.2.1),
- CRO3 - $\uplus^{73}$ community revision operator, the maximal admissible depth of the clause derivation equals to 3,
- CRO4 - $\uplus^{90}$ community revision operator, the maximal admissible depth of the clause derivation equals to 4,
- CRO5 - $\uplus^{235}$ community revision operator, the maximal admissible depth of the clause derivation equals to 5.

The number of the generated states decreases with an amount of the clauses revised within the **revise***time* due to more knowledge about the object agents. This additional knowledge is generated within the **revise***time*. See graph 5.4 for the number of the generated clauses within the **revise***time*. The number of the generated states increases with an amount of the clauses revised within the **revise***time*.

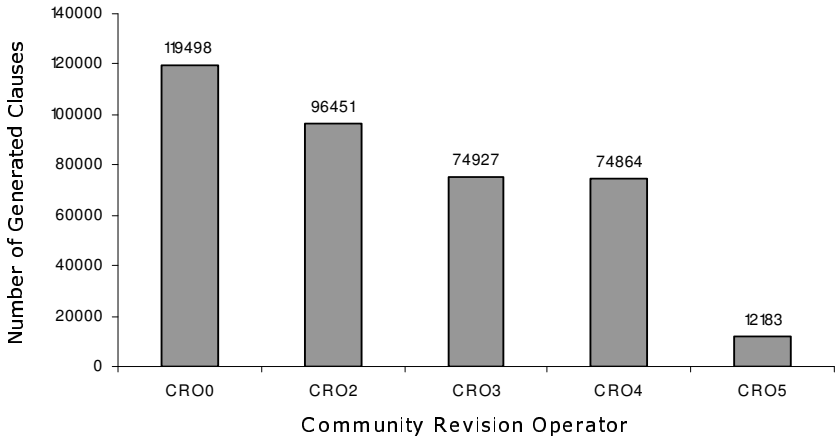

Fig. 5.3. Number of Generated Clauses within the **inspect***time*.

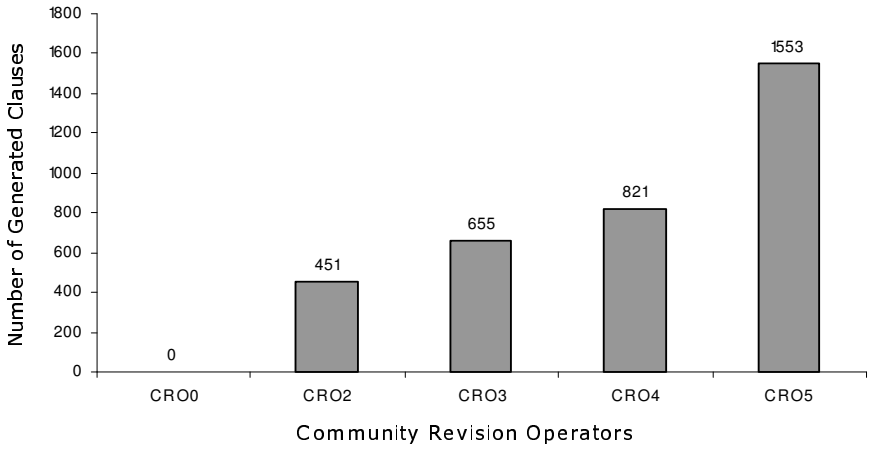

Fig. 5.4. Number of Generated Clauses within the **revise***time*.

Let us consider the number of the generated clauses within the **inspect***time* and the **revise***time*. We sum up the number of the generated clauses per proof within the **inspect***time* and the number of the generated clauses within the **revise***time*. We are interested if the deduction of new knowledge within the **revise***time* is beneficial for the number of the generated clauses within only one proof in the **inspect***time*. See graph 5.5 for the result. The number of the generated clauses decreases with an amount of the clauses revised within the **revise***time*. This is due to the shortening strategy (see section 4.2.1) applied within the **revise***time*. The shortening strategy prevents creation of huge amount of the generated clauses within the **revise***time* via the strategy

to generate clauses shorter than one of their parents. There is more generated clauses with the CRO4 than with the CRO3. The difference between the generated clauses within the **revise***time* by CRO4 and CRO3 is higher than difference between the generated clauses within the **inspect***time*.

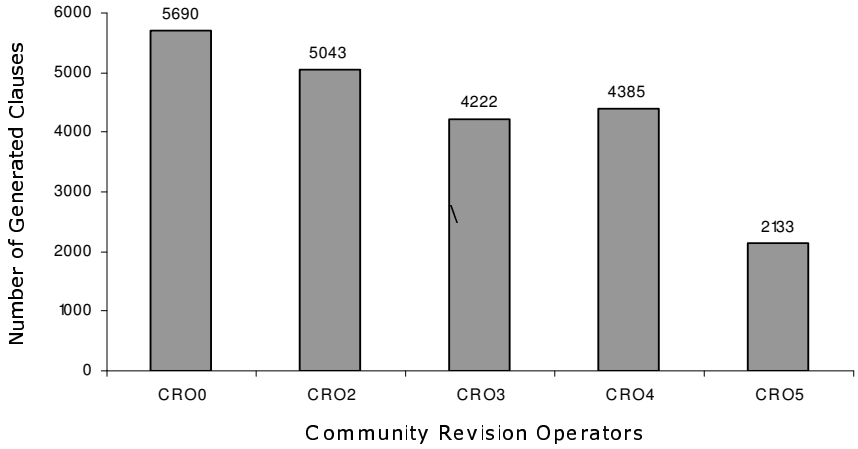

Fig. 5.5. Total Number of Generated Clauses within the **inspect***time* and the **revise***time*.

See graph 5.6 for the time responses within the **inspect***time*. The time necessary for responses decreases with an amount of the clauses revised within the **revise***time*. The time responses do not depend on the number of the generated states. The time responses of the CRO4 community revision operator are a little higher than the time responses of the CRO3 community revision operator although the number of the generated states is lower. It is necessary to be aware of the fact that we have to consider an operation performed with every generated clause. With growing amount of knowledge there grows complexity of the operations performed with every generated clause, for example subsumption process. Then the time responses can grow although the number of the generated states is decreased (see following experiment). The time responses with the CRO3 operator are very low due to the fact that most of the clauses from the gbb are revised within the **revise***time* - it is an extreme operator and the creation of the clauses revised within the **revise***time* takes a lot of the time.

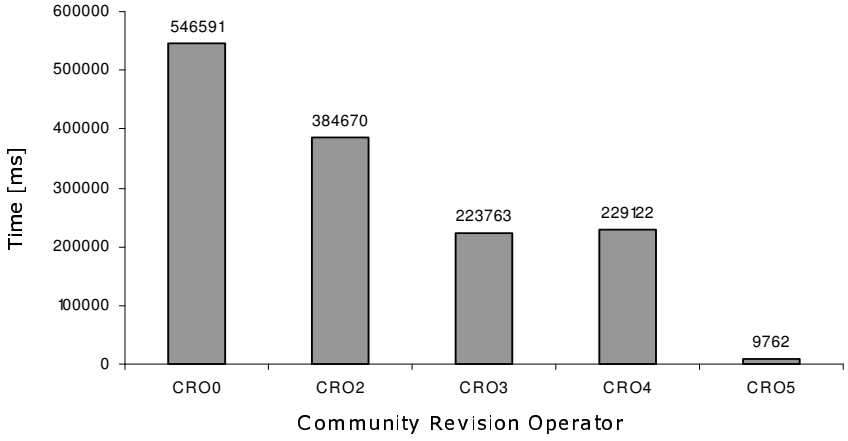

Fig. 5.6. Time Responses within the **inspect***time*.

5.4 Utilization of Domain Knowledge in inspect*time*

Let us define now the community inspection phase in such ways:

$$\begin{aligned} &\mathsf{ebb}_t(\theta) \cup \mathsf{gbb} \vdash \mathtt{goal}_t \ \textit{or} \\ &\mathsf{ebb}_t(\theta) \cup \mathsf{gbb} \cup \mathsf{event\_lemmas}_t(\theta) \vdash \mathtt{goal}_t. \end{aligned} \tag{5.1}$$

In the contrast of classical theorem proving task we are able to separate the axioms submitted to the theorem proving task into the disjunctive parts. The time responses depends on the number of the generated clauses and on the operations performed with every generated clauses. Our attempt is to reduce the `CLOSE` list used by the searching algorithm for forward subsumption process within the theorem proving process. We removed $\mathsf{ebb}_t(\theta)$ from the `CLOSE` list and we applied both ways of the community inspection phase defined in equation 5.1. See the graph 5.7 for the number of the generated clauses and the graph 5.8 for the time responses within the **inspect***time* when the first theorem proving process in equation 5.1 is used.

We found that the number of the generated clauses has not been changed in both cases. Be aware that it need not to be always true. We reduce the time responses in both cases due to reduction of the operations performed with every generated clauses.

We experimented with different settings of the `CLOSE` list. The `CLOSE` list was either set as an empty after the initialization of the theorem proving task or set to $\mathsf{event\_lemmas}_t(\theta)$ only. We reduce the time responses in both cases although the number of the generated clauses is increased in both cases. The time responses were lower than in the case of previous setting of the `CLOSE`

list due to the fact explained above. The number of the generated clauses was higher because the subsumption process was not such efficient.

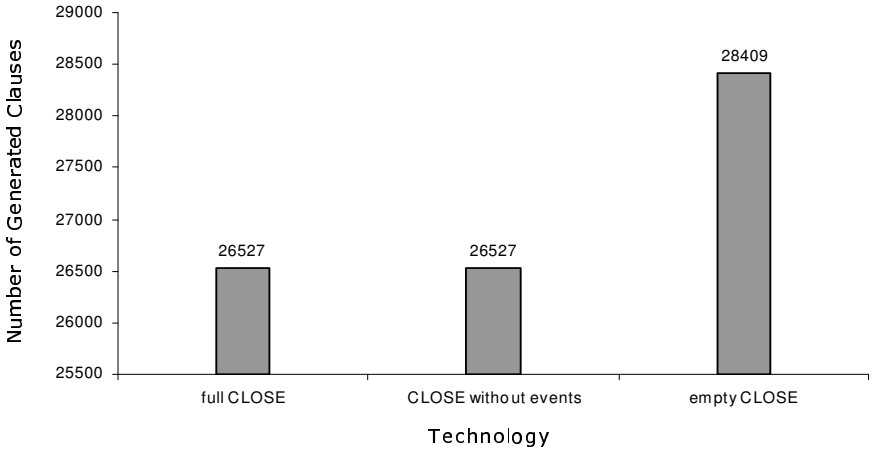

Fig. 5.7. Number of Generated Clauses within the **inspect***time*.

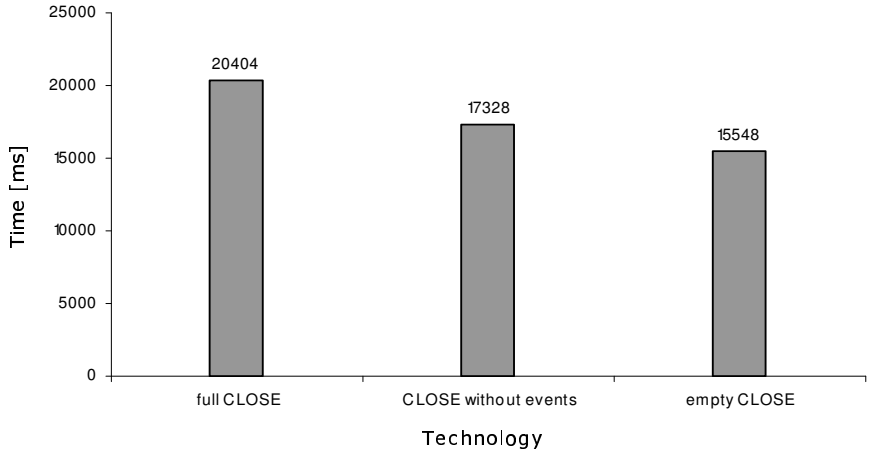

Fig. 5.8. Time Responses within the **inspect***time*.

5.5 Model Elimination and Literal Preference

We have combined set of support strategy (below denoted as `SOS`) with model elimination method for the theorem proving process within the **inspect***time* (see section 4.2.2). This strategy is called `SOS-ME`. We compare a pure `SOS` strategy with `SOS-ME` strategy.

Experiments have confirmed the reduction of the number of the generated clauses and the reduction of the time responses within the **inspect***time* if we use `SOS-ME` strategy. For an explanation of the reductions see 4.2.2.

We have expanded `SOS-ME` strategy by the ability to choose the literal in the clause which will be eliminated (see section 4.2.2). This feature is called a **literal preference**. Experiments have confirmed further reduction of the number of the generated clauses and reduction of the time responses within the **inspect***time*. For an explanation of the reductions see 4.2.2.

See the graph 5.9 for the number of the generated clauses and the graph 5.10 for the time responses within the **inspect***time*. We used technologies:

- `SOS` – `SOS` strategy,
- `SOS-ME` – `SOS-ME` strategy and
- `SOS-ME, LP` – `SOS-ME` strategy supported by the literal preference.

Time limit of 210 seconds has been set, the computation was interrupted then. `SOS-ME` strategy supported by the literal preference succeeded to derive all queries, while each of remaining technologies was not successful in some queries.

We proved within the experiments that by implementation of model elimination and the literal preference methods we improved **revise***time* in the sense of the number of the generated clauses and of the time responses.

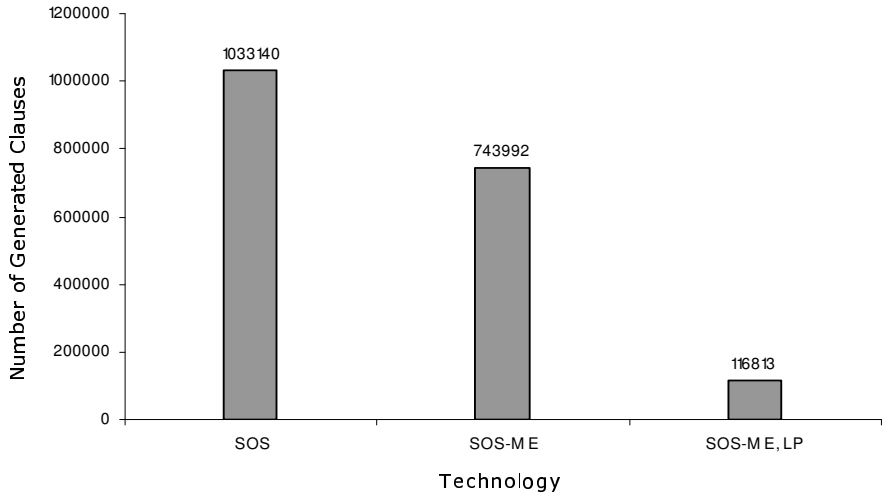

Fig. 5.9. Number of Generated Clauses within the **inspect***time*.

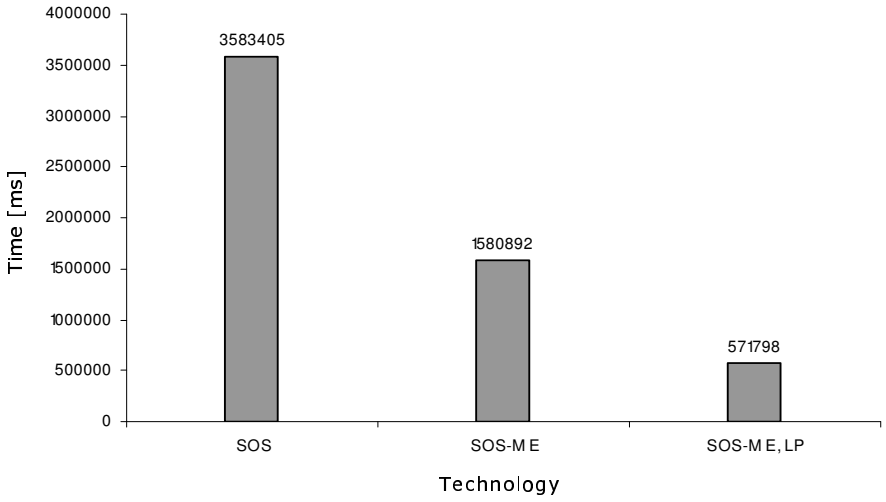

Fig. 5.10. Time Responses within the **inspect***time*.

5.6 Proof Lemmas, Caching, Lemmatizing, Deletion with Lemmas

We proposed the lemmas generation as a side effect of the theorem proving process within the **inspect***time* (see section 4.2.2). Proof lemmas are ground unit clauses (containing only the constants) in most cases in our domain. We used three technologies how the proof lemmas are applied within the same proof where they are generated:

- *deletion with lemmas* method,
- *lemmatizing* and
- *caching*.

We compared the technologies' impact to the theorem proving process within the experiments. The theorem proving process has been supported by $\uplus^{\min}$ (no deduction within **revise***time*) and by various combination of considered technologies, namely: `SOS-ME` strategy, the literal preference, generation of the proof lemmas, deletion with lemmas method, caching, lemmatizing. See the graph 5.11 for the number of the generated clauses and the graph 5.12 for the time responses within the **inspect***time*. The technologies are denoted in the graphs in a such way:

- `No Lemma` – the generation of the proof lemmas is not used,
- `DwL` – deletion with lemmas method,
- `C` – caching and
- `L` – lemmatizing.

Deletion with lemmas method always reduces the number of the generated clauses due to the ability to remove the tautologies. The reduction is very strong. Effectiveness of *deletion with lemmas* method is supported by the fact that our domain is characterized by high number of constants in literals. Then the subsumption process is able to remove more tautologies with respect to the proof lemmas represented by ground facts in most cases. *Caching* always reduces the number of the generated clauses too. The reduction is light because the number of successful applications of *caching* is low. *Lemmatizing* increases the number of the generated clauses in some cases. When $\texttt{goal}_t$ contains variables then *lemmatizing* increases the number of the generated clauses because the unification process supported by the proof lemmas is more successful. The combination of the technologies (except the combination of the *deletion with lemmas* method and *caching* - there is light reduction) increases the number of the generated clauses with respect to the number of the clauses generated by the proof supported by the *deletion with lemmas* method only.

Deletion with lemmas method reduces the time responses due to the reduction of the number of the generated states. The resulting reduction is very strong. The *lemmatizing* reduces the time responses too. This is probably due to the fact that the system chooses for expansion process the clauses from which it was possible to derive empty clause. The reduction is not as

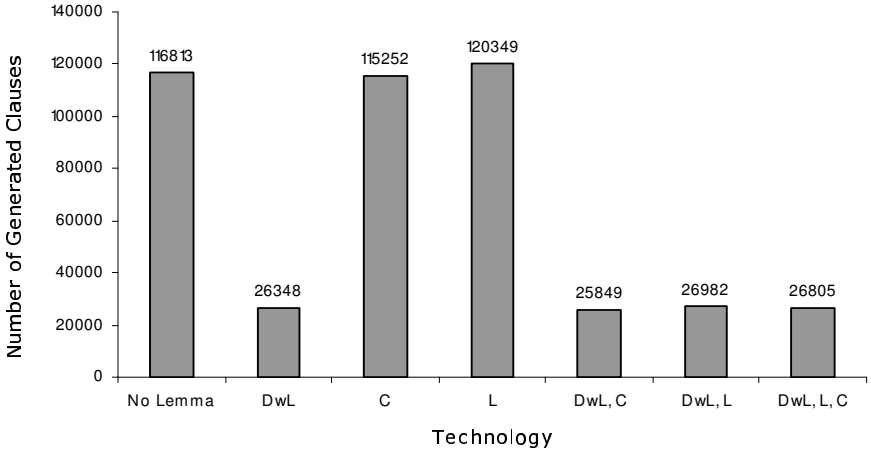

Fig. 5.11. Number of Generated Clauses within the **inspect***time*.

strong as in the case of using *deletion with lemmas* method. The number of the clauses expanded and moved to `SOS` list (see section 4.2.2) is lower than in the case of using *caching*. Be aware of the fact that this claim depends on the considered domain, consequently it need not to be true in different domains (`gbb`). *Caching* reduces the time responses due to the reduction of the number of the generated states. The reduction is light. Combination of the *deletion with lemmas* and *lemmatizing* decreases the time responses with comparison of using isolated methods. Combination of the *Deletion with lemmas* method and *caching* increases the time responses with comparison of using *deletion with lemmas* method only. The reason is the fact that the light reduction of the time responses by the *caching* is not so strong to cover the requirements increased by the operations performed over the `USABLE` list (see section 4.2.2), which is expanded about the proof lemmas. Adding *caching* to the *deletion with lemmas* method and *lemmatizing* decreases the time responses. The additional operations over the `USABLE` list are already included by using of *lemmatizing*. Now *caching* reduce the number of the generated clauses then it reduce a little the time responses. Successful application of *caching* is again low.

With respect to our experiments we recommend the most effective technology: *deletion with lemmas* method. Supporting of *deletion with lemmas* method by *caching* and *lemmatizing* can increase the number of the generated clauses and consequently the time responses. *Deletion with lemmas* method cannot increase the number of the generated clauses. In the worst case *deletion with lemmas* method increases slightly the time responses because the requirements of the subsumption process are increased. We found some such queries within the experiments. But for us it is more important that

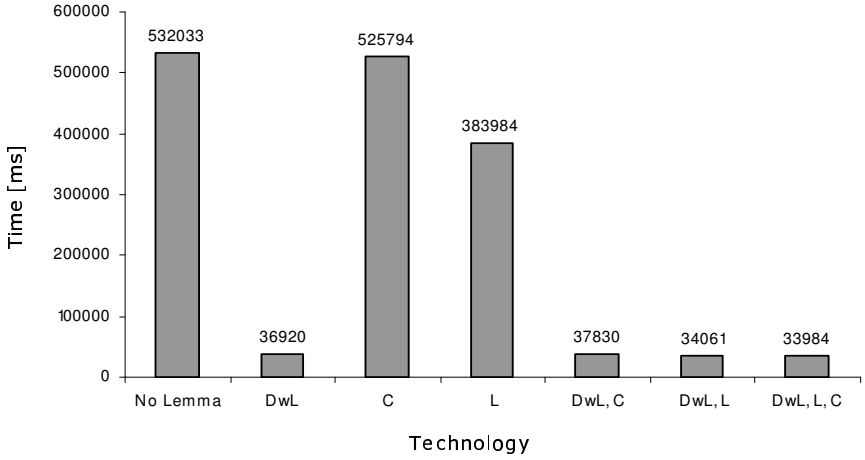

Fig. 5.12. Time Responses within the **inspect***time*.

by using of *deletion with lemmas* method we reduce minutes or seconds for some queries than that we add milliseconds for some queries. *Caching* is careful technology which negligible influence. While *lemmatizing* is more radical technology which either more decreases or more increases the time responses of the theorem proving process.

5.7 Event Lemmas, Caching, Lemmatizing, Deletion with Lemmas

We can make similar conclusion about the technologies as in the section 5.6. Using the event lemmas is not so efficient as using the proof lemmas. Proof lemmas are generated within the specific proof and the probability of their using by any technology is higher than using of the event lemmas which are deduced within **revise***time* without the relation to the future $\texttt{goal}_t$. That is the reason why the *deletion with lemmas* method and *lemmatizing* have generated more clauses than in the case of using the proof lemmas. The *caching* has not changed with respect to *caching* supported by the proof lemmas. The combination of the technologies decreases somewhat the number of the generated clauses with respect to the proof supported by *deletion with lemmas* method only. See the graph 5.13 for the number of the generated clauses and the graph 5.14 for the time responses within the **inspect***time*. The technologies are denoted in the same way as in the section 5.6.

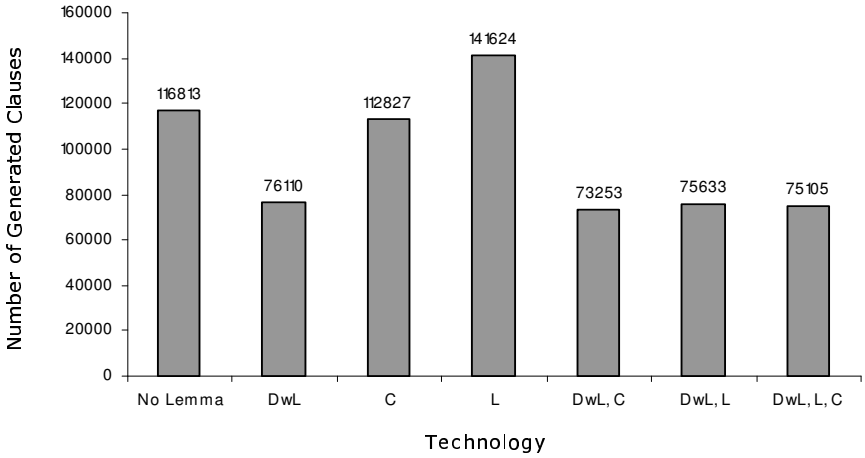

Fig. 5.13. Number of Generated Clauses within the **inspect***time*.

Deletion with lemmas method decreases the time responses. The reduction is strong but not as high as in the case of the proof supported by the proof lemmas. *Lemmatizing* increases the time responses here because the number of the generated clauses is higher than in the case of of the proof supported by the proof lemmas. The *caching* has not changed with respect to *caching* supported by the proof lemmas. The combination of the technologies decreases a little the time responses with respect to the proof supported by *deletion with lemmas* method only.

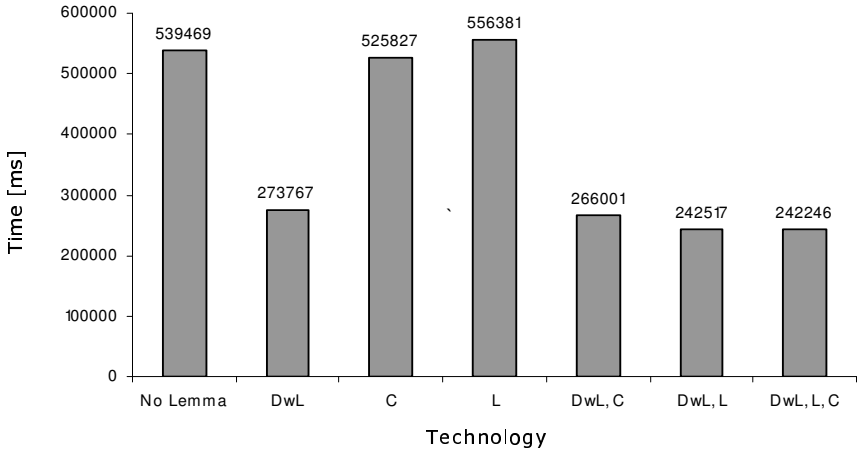

Fig. 5.14. Time Responses within the **inspect***time*.

As we mentioned above using the event lemmas is not as efficient as using the proof lemmas. That is why we will prefer the proof lemmas to the event lemmas.

5.8 Proof Lemmas, Event Lemmas

We found within the experiments that using of the proof lemmas is more efficient than using of the event lemmas (see section 5.7). We were interested if by appending of the event lemmas to the proof lemmas can we improve the **inspect***time* in the sense of the time responses. We used the event lemmas generated by the $\uplus^{90}$ community revision operator. We reduce the number of the generated states by appending the event lemmas to the proof lemmas due to the using of more knowledge by the technologies *deletion with lemmas* method and *lemmatizing* with *caching*. The reduction is light. On the other hand the time responses are increased because the forward subsumption process took a lot of the time due to the long `CLOSE` list (see section 4.2.2). The `CLOSE` list contains both event and proof lemmas. Unrestricted appending of the event lemmas does not improve the **inspect***time* because the event lemmas consist of huge amount of knowledge which is not related to $\texttt{goal}_t$.

We experimented with appending of the filtered event lemmas with respect to the actual $\texttt{goal}_t$. We observe reduction of the number of the generated clauses with respect to the proof supported by the proof lemmas only and to the proof supported by the proof lemmas and the unfiltered event lemmas. This is due to more effective utilization of the technologies *deletion with lemmas*, *caching* and *lemmatizing* (more knowledge is available) and the *lemmatizing* generates lower number of the clauses than in the case of using the unfiltered event lemmas. We reduce the time responses with respect to the proof supported by the proof lemmas only and to the proof supported by the proof lemmas and the unfiltered event lemmas. This is due to the reduction of the number of the generated clauses and to the reduction of the number of used the event lemmas placed to the list `CLOSE`.

See the graph 5.15 for the number of the generated clauses and the graph 5.16 for the time responses within the **inspect***time*.

Experiments confirm that unrestricted using of the event lemmas does not improve the **inspect***time* with respect to the using of the proof lemmas. The event lemmas consist of huge amount of knowledge which is not related to $\texttt{goal}_t$. It is necessary to filter the event lemmas with respect to $\texttt{goal}_t$.

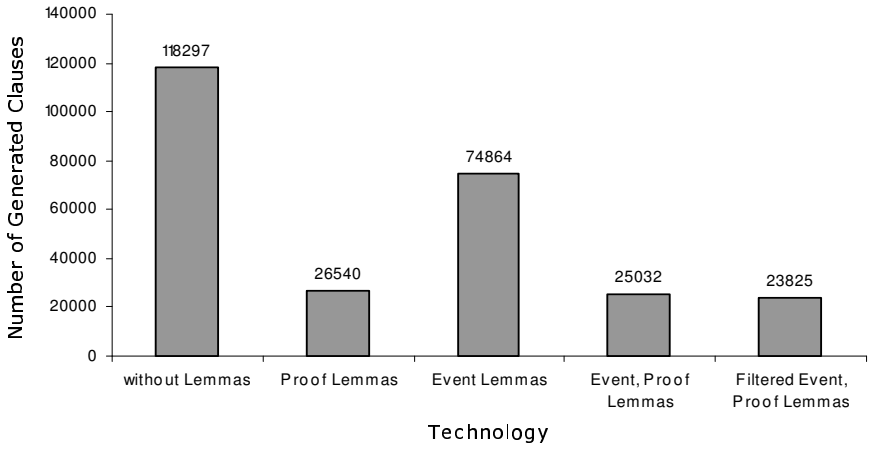

Fig. 5.15. Number of Generated Clauses within the **inspect***time*.

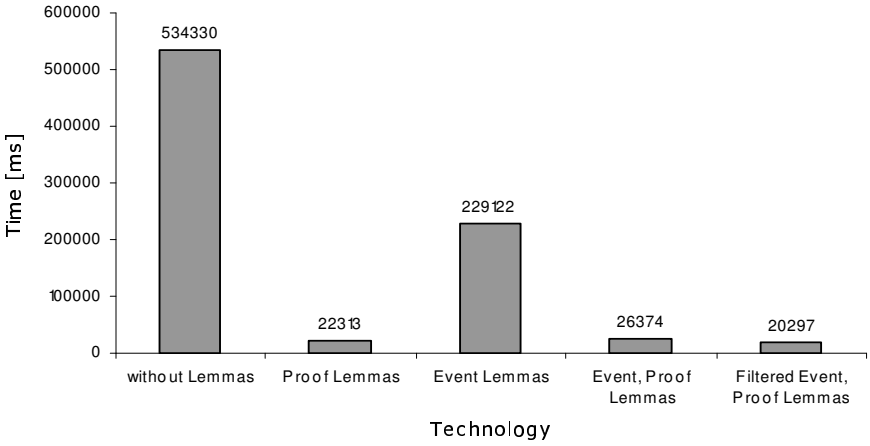

Fig. 5.16. Time Responses within the **inspect***time*.

5.9 Comparison with OTTER Theorem Prover

We compared our theorem proving results with another theorem prover. We have chosen `OTTER` theorem prover [21] based on `SOS` strategy. Our theorem proving process is supported by $\uplus^{\min}$ (no deduction within **inspect***time*), `SOS-ME` strategy, literal preference, generation of the proof lemmas, *caching*, *lemmatizing* and *deletion with lemmas* method. The proofs has been limited by 180 seconds. See the graph 5.17 for the number of the generated clauses and the graph 5.18 for the time responses within the **inspect***time*. The queries we

sorted with respect to the ratio between the variables and constants in $\texttt{goal}_t$ (0 – no variable, 1 – no constant).

Most often, our theorem prover generates less clauses than `OTTER`. There are several queries when `OTTER` generates lower the number of the generated clauses due to used UR-resolution and hyperresolution implemented within the prover.

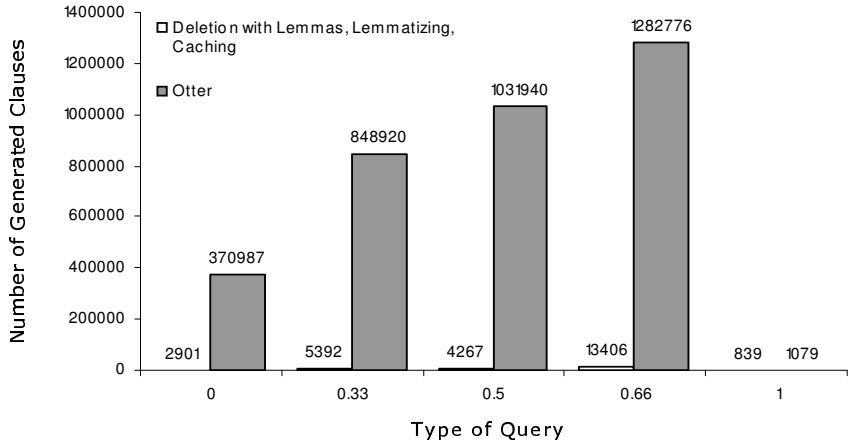

Fig. 5.17. Number of Generated Clauses within the **inspect***time*.

Several observation holds for the time responses. There are several queries when `OTTER` is faster due to used UR-resolution and hyperresolution and to the fact that `OTTER` is implemented in `C` programming language. This programming language is much faster than `Lisp` programming language used for our theorem prover. There are just a few queries when `OTTER` is not able to reply within 180 seconds.

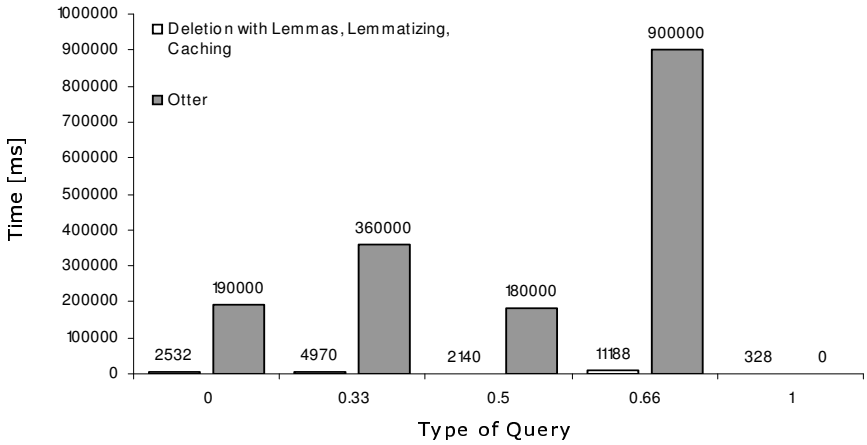

Fig. 5.18. Time Responses within the **inspect***time*.

5.10 Conclusion

During the last years, we have gained considerable experience in design and utilization of agents social knowledge about its environment e.g. for efficient coalition formation in CPlanT coalition planning system. Social knowledge proved useful for this purpose provided that it offers true up-to-date information about the agent's environment and about the other agents in the community. The social knowledge can be used not only for coalition formation but it plays a role in any reasoning the agent has to perform. In relation to its environment and to the events arising in it. This is not a rare activity - there are many reasons why an agent in a MAS system has to reason. Let us review just a few of them:

1. The environment is so complex that it is impossible to review all its properties in advance, instead the agent has to attempt "understanding" or "making sense" of available information on-line as it arrives.
2. The properties of the environment are not fixed, but they are evolving/changing during the life span of the agent - the agent has to identify the changes by itself.
3. The agents in MAS have some private knowledge which they do not make public. It consists of the rules which govern their behavior - they are not public but their knowledge is important for proper decision-making during the coalition formation phase of MAS behavior.

This report is concerned with the last case first of all, but the techniques designed here can be applied in the other cases as well. Each agent maintains its own model of its partners in MAS - this model is supposed to explain the considered agents' behavior. In an ideal case agent's behavior is a consequence

of this model. The agent can reason about its percepts as well as about information it is getting from the other members of the system. If the agent has simultaneous access to information gathered by several agents, it performs meta-reasoning. Whenever an agent obtains some new information concerning agent A, it revises its model of the agent A so that the resulting model of A is consistent with all observations. We have defined several approaches how to ensure such revision. We distinguish the following three core different strategies for updating the model:

- the simplest one adds to the existing model the observed information and nothing more is performed,
- the deductive extension enhances the existing model by consequences of the considered observation,
- the inductive extension uses machine learning techniques to generate a new model which is consistent with all information gathered about the agent A up to now.

The agent uses its actual model of agent A whenever it needs to know how the agent A will react to some request - the phase is called the inspection phase. Complexity or time required for the inspection phase is not fixed - of course, it is influenced by the approach applied in the revision phase.

We have implemented several different operators ensuring various versions of model-updating as well as alternative solutions performing activities corresponding to the inspection phase. *The deductive part is based on the resolution principle enhanced by some heuristics (see section 4.2), the inductive part uses two techniques of machine learning, namely version space (section 4.3.1) and ILP (section 4.3.2).* The performance of the implemented operators has been compared and tested in an experimental coalition planning system – CPlanT.

Our experiments (in chapter 5) verify that the suggested approach to meta-reasoning is feasible. The agent can utilize its experience to build a model of its peer agents which offers considerable advantage when used in its subsequent decision making. The complexity of the deductive revision using automated reasoning depends on the choice of used heuristics influencing number of generated clauses. If more complex revision is used the inspection process is faster. It seems that proper compromise ensuring optimal performance has to be designed for each specific application. An interesting research question remains open: *Is it possible to establish a relation between syntactic properties of the language used for description of the agent model and good choice of these parameters?*

Automated reasoning seems to be an appropriate methodology for implementation of meta-reasoning capabilities. We experimented with lemmas generated within the **revise***time* and the **inspect***time*. We recommend the technology *deletion with lemmas* method supported by the lemmas generated within the **inspect***time* which reduces the time necessary for the responses in the **inspect***time*. The lemmas generated within the **revise***time* are not so efficient.

Interesting results are due to comparison of performance between deductive and inductive approach (see section 5.3). Inductive approach differs from the former one by the fact that its applies generalization. The resulting revised model has to be understood as a hypothesis only. This hypothesis is able to answer some queries which have not been possed to the agent before. This might lead to errors sometimes. Success rate of induction based on ILP proved to be very good. The prise for this is slow-down in performance when compared with version space based solution. It seems that in some environments both approaches could be combined - version space approach could be used in time critical periods while in more relaxed periods ILP approach could be applied. ILP is a very good choice in domains characterized by complex background knowledge. This happens e.g. whenever timing of the events has to be taken into account (information about time and its ordering has to be described) and the rules of behavior depend on sequencing of events. Such a behavior can be best described in the language of 1^{st} order logic. Both deductive and inductive approaches to agent model revision proved most promising and our experiments point to interesting new research directions which will lead to enhancement of the systems performance and to creation of more precise guidelines when to use which of the mentioned approaches.

5.11 Comments on Project Results

This section is supposed to evaluate the project results form the point of view of the project white-paper from 15-May-2002.

The work has been organized (as planned in the white-paper) into four mutually dependent workpackages:

1. Domain analysis (month 1 – month 2): Here we have analyzed the role of the meta-reasoning process in multi-agent systems. This analysis has been carried out in a general way, but also with respect to the domain of planning of humanitarian relief operations. The abstract architecture for meta-reasoning in multi-agent systems has been designed and published in:

   Pěchouček, M., Štěpánková, O., Maříık, V. and Bárta, J. Abstract Architecture for Meta-reasoning in Multi-agent Systems. In: Maříık V., Mueller J., Pěchouček M. (Eds.) **Multi-Agent Systems and Applications III.** *Lecture Notes in Computer Science. 2691* Heidelberg : Springer, 2003, p. 85-99. ISBN 3-540-40450-3.

   Tožička J., Bárta J. and Michal Pěchouček. Meta-Reasoning for Agents' Private Knowledge Detection. In Klusch, M., Ossowski, S.,Omicini, A., Laamanen, H. (Eds.) **Cooperative Information Agent VII**, *Lecture Notes in Computer Science, LNAI 2782* Heidelberg : Springer, 2003. ISBN 3-540-40798-7

   It has been decided to test the functionalities of the implemented meta-reasoning on the CPlanT multi-agent system. The agents in the system

perform cooperative and self-interested behaviour, however do not perform any form of adversarial behaviour.

2. Monitoring/knowledge acquisition (month 3 – month 6): Here we have concentrated on developing robust monitoring capability with the CPlanT system. A powerful **subscription based mechanism** (see section 3) for monitoring agent's interaction in purely collaborative environment. For the self-interested environment we have designed and developed a simple **intruder agent** (a realization of the concept of sniffing-agent and the chameleon-agent) (see section 3.3) that sniff the information about agent interaction and makes this available to the meta-agent.
3. Knowledge analysis/community revision (month 6 – month 9): This phase was the key to providing novel research results to the community. We have tested three different approaches to analyzing the knowledge that were made available by monitoring subsytem: **simulation**, **deductive reasoning** (implemented by classical theorem proving technology), and **inductive reasoning** (implemented by Version Space and Inductive Logic Programming machine learning techniques). All four methods have been implemented and all but ILP have been integrated into the body of the meta-agent. The prime objective of meta-reasoning has been to predict agents behaviour. Little effort has been devoted to studying on an impact of the object level agents after having the information available (community revision phase).
4. Implementation and testing (month 5 – month 12): Within this part of the research project we have tested meta-reasoning methods and experimented with different settings and operation modes of meta-reasoning (see chapters 5 and 5.10).

5.12 Open Questions

The results of the research project met expectation of the researchers and answered many of the asked questions. However, given the rather wide scope of the project and limited amount of time, there are several challenging issues still lacking deeper investigation:

- **adversarial domain:** In the test domain of the CPlanT system we assume all agents to be truth-telling, they do not change their private knowledge over the time and more-over, we do not consider an adversarial agent that could harm the object agent (such as blocking resources, carrying out dos attack on selected services, providing misleading information). It would be interesting to see how the meta-agent operates in the adversarial domain.
- **adaptivity:** There was little time to investigate the concept of the community revision within the project. It would be interesting to investigate

and measure an impact of availability of the knowledge provided by meta-reasoning capability to the object agents (e.g. increased efficiency, improved security, ...). This lead to the question of how much the agents will be happy to be monitored given the added value of their improved performance.

- **cooperation of methods:** We have worked with and investigated meta-reasoning methods separately. However, preliminary testing shows that different approaches are suitable for different sets of data. In more complex systems an autonomous cooperation of different meta-reasoning methods may be interesting to investigate.
- **peer-to-peer meta-reasoning :** The concept of meta-agent is powerful as it allows higher forms of reasoning without any direct burden on agents computational resources. However, especially in the adversarial environment the agents may want to be capable of meta-reasoning on the object level and improve their operation in the hostile environment without subscribing to a central (while independent) meta-agent. Meta-reasoning in adversarial environment must be taken down to the agents peer-to-peer interaction.

5.13 Acknowledgement

The project work has been co-funded by European Office for Aerospace Research and Development (EORD) Air Force Research Laboratory (AFRL) – contract number: FA8655-02-M-4056, Office of Naval Research (ONR) – award number: N00014-03-1-0292 and by the Grant No. LN00B096 of the Ministry of Education, Youth and Sports of the Czech Republic.

A

Language for Describing Object Agents' Behavior

In this part of the appendix we describe the decision making algorithm used by the object agents within the team allocation process.

A.1 Object Agents' Decision Making Algorithm

Meta-reasoning activity focuses on the most important part of the object agents' private knowledge - the team restrictions. We could design teams, coalitions and resource allocation for the disasters and missions if we would be aware of this restriction type. The team restriction consists of the parts:

- restriction specifying the maximal number of the team members,
- restriction specifying non-acceptable team leaders,
- restriction specifying the locations, where the object agent does not participate.

See section **??** for definition of the task (it consists of the proposed team, team leader and location).

A.1.1 Restriction Specifying Maximal Number of Team Members

The object agents accept teams up to specific number of the team members. It creates all teams, which it would accept, within an alliance formation process with respect to the restriction specifying the maximal number of the team members. The object agent refuses the task within the team allocation request if it is not aware of the team proposed within specific task.

A.1.2 Restriction Specifying Non-acceptable Team Leaders

We used three the object agent's public attributes in order to express the restriction:

- the object agent's country location, like "Suffer Terra", "Country-A",
- the object agent's city location, like "Suffer Town", "Sunny Side Port" and
- the object agents's type, like "Government", "Religious".

These attributes with their values are joined into the leader restriction via logical connector *AND*. There can be defined several leader restrictions ("or" relation between simple restrictions), see the example A.1.2.

There are leader restrictions:

- $(city\ "SufferTown") \wedge (country\ "SufferTerra")$ and
- $(country\ "Country-B") \wedge (type\ "Government")$.

All object agents either from the country "Suffer Town", city "Suffer Town" or from the country "Country-B" with type "Government" will be refuted.

The object agent:

- refuses the task within the team allocation request proposed by another object agent - team leader, where team leader's description meets with any team restriction and
- proposes no team parameters within the team proposal to coalition leader, where coalition leader's description meets with any team restriction.

A.1.3 Restriction Specifying Non-acceptable Locations

To every city in suferterra scenario [37] is assigned the population composition, see the example A.1.3.

The city "Suffer Town" consists of:

1. 65% of atheist,
2. 20% of christian,
3. 7% of muslim,
4. 4% of native and
5. 4% of other population.

The object agent may have defined one or more ("or" relation between simple restrictions) population restrictions, for example: *native - 80*. There are cities (locations) which have 80% or higher native people. The object agent:

- refuses the task within the team allocation request invoked in order to participate in locations with population equal or higher than any the object agent's population restriction and

- does not propose its services within the team proposal if it proposes the team allocated to participate in locations with any population equal or higher than any the object agent's population restriction.

A.2 Object Agents' Decision Making Algorithm Formally

Let us describe the object agent's decision making more formally. We use the first order logic without equality and function symbols. We utilize the relation operator $<$. First we express the object agent's behavior - the set denoted as t_{beh}:

If the object agent accepts the task (team allocation request), then it accepts the team (number of team members), team leader and location defined within the task. If the object agent refuses the task then it refuses minimal at least one of the task parameter:

$\forall L \forall C \forall N accept(L, C, N) \Leftrightarrow$
$accept\_city(C) \wedge accept\_leader(L) \wedge accept\_number(N).$

If the object agent accepts specific number of the team members then it will accept lower number of the team members. If the object agent refuses specific number of the team members then it refuses higher number of the team members:

$\forall M \forall N accept\_number(N) \wedge \ M < N \Rightarrow accept\_number(M).$

If the object agent accepts the population number then it accepts lower population number. If the object agent refuses the population number then it refuses higher population number:

$accept\_population(P, N) \wedge \ M < N \Rightarrow accept\_population(P, M).$

If the object agent accepts specific location then it accepts all location's population numbers. If the object agent refuses specific location then it refuses at least one of the location's population number. Following formulae are defined with respect to the sufferterra scenario [37].

$accept\_population(\text{"}ATHEIST\text{"}, 65)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 20)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 7)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 4)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 4)) \Leftrightarrow accept\_city(\text{"}Suffer\ Town\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 54)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 23)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 3)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 1)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 9)) \Leftrightarrow accept\_city(\text{"}Central\ Town\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 30)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 8)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 60)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 1)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 1)) \Leftrightarrow accept\_city(\text{"}Sunny\ Side\ Port\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 70)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 15)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 1)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 10)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 4)) \Leftrightarrow accept\_city(\text{"}Central\ Lake\ City\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 68)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 16)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 1)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 11)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 4)) \Leftrightarrow accept\_city(\text{"}St.\ Josephburgh\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 70)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 13)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 0)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 15)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 2)) \Leftrightarrow accept\_city(\text{"}North\ Port\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 12)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 3)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 0)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 85)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 0)) \Leftrightarrow accept\_city(\text{"}Coast\ Village\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 0)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 0)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 0)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 100)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 0)) \Leftrightarrow accept\_city(\text{"}Sunset\ Point\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 75)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 24)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 1)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 0)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 0)) \Leftrightarrow accept\_city(\text{"}Town - A\text{"}),$

$accept\_population(\text{"}ATHEIST\text{"}, 75)$
$\wedge\ accept\_population(\text{"}CHRISTIAN\text{"}, 25)$
$\wedge\ accept\_population(\text{"}MUSLIM\text{"}, 0)$
$\wedge\ accept\_population(\text{"}NATIVE\text{"}, 0)$
$\wedge\ accept\_population(\text{"}OTHER\text{"}, 0)) \Leftrightarrow accept\_city(\text{"}City - A\text{"}),$

$accept\_population("ATHEIST", 70)$
$\wedge\ accept\_population("CHRISTIAN", 20)$
$\wedge\ accept\_population("MUSLIM", 8)$
$\wedge\ accept\_population("NATIVE", 0)$
$\wedge\ accept\_population("OTHER", 2)) \Leftrightarrow accept\_city("Town - B"),$

$accept\_population("ATHEIST", 78)$
$\wedge\ accept\_population("CHRISTIAN", 20)$
$\wedge\ accept\_population("MUSLIM", 1)$
$\wedge\ accept\_population("NATIVE", 0)$
$\wedge\ accept\_population("OTHER", 1)) \Leftrightarrow accept\_city("City - B"),$

$accept\_population("ATHEIST", 30)$
$\wedge\ accept\_population("CHRISTIAN", 0)$
$\wedge\ accept\_population("MUSLIM", 65)$
$\wedge\ accept\_population("NATIVE", 0)$
$\wedge\ accept\_population("OTHER", 5)) \Leftrightarrow accept\_city("Town - C").$

We append formulae about possible the object agent's restrictions - the set denoted as t_{res}. We suppose one specifying maximal number of the team members and specific population (more restriction on one type has no sense). If there is a restriction specifying the maximal number of the team members for example x, $y = x + 1$ then the formulae are appended:

$accept\_number(x),$

$\neg accept\_number(y).$

If there is no restriction specifying the maximal number of the team members then the formula is appended:

$\forall X accept\_number(X),$

If there is a restriction specifying the population number of specific type *type* for example x,$y = x - 1$ then the formulae are appended:

$accept\_population(type, y),$

$\neg accept\_population(type, y).$

If there is no restriction specifying the population number of specific type *type* the formula is appended:

$\forall X accept\_population(type, X),$

Let us focus now on the restrictions specifying non-acceptable team leaders. We transform the team leader restrictions to DNF form. We utilize attributes described in section A.1.2 as literal names with two parameters: restricted value equivalent to the attribute type and variable equivalent to the object agent (team leader). One team leader restriction is defined via *AND*

logical connector over literals equivalent to the attributes. Full leader restriction is defined via *OR* logical connector over simple leader restrictions. Let us denote the team restriction as $leader\_restriction(L)$, where L is the team leader (see the example A.2).

The leader restrictions from the example A.1.2 is transformed to the form: $(city(L,"SufferTown") \wedge country(L,"SufferTerra")) \vee (country(L,"Country-B") \wedge type(L,"Government"))$.

If $leader\_restriction(L) \neq nil$ we append the formula:

$\forall L \; \neg accept\_leader(L) \Leftrightarrow leader\_restriction(L)$,

else we append the formula:

$\forall L \; accept\_leader(L)$.

We append knowledge [1] about all the A object agent's cooperators in the system - all object agents except A - let as denote them as $\varepsilon(A)$ [1]. Let us define at_s as a set of the attributes (at_i) for the object agent's description, we suppose that it is a finite set:

$$at\_s = \{at_i\}. \tag{A.1}$$

The attribute at_i have several possible values, we suppose that it is a finite set:

$$\forall at_i \in at\_s : at_i = \{value_j\}. \tag{A.2}$$

We append facts about A object agent's cooperators according to the algorithm:

$$\begin{array}{l}
\forall A_i \in \varepsilon(A) : \\
\quad \forall at_i \in at_s : \\
\qquad actual_value = get_value(A_i, at_i); \\
\qquad \forall value_j \in at_i : \\
\qquad\quad if \; actual_value == value_j \\
\qquad\qquad append_fact(at_i(A_i, value_j)); \\
\qquad\quad else \\
\qquad\qquad \neg append_fact(at_i(A_i, value_j));
\end{array} \tag{A.3}$$

The object agent accepts the task within the team allocation request only if:

---

[1] It is done only if $leader_restriction(L) \neq nil$

$$t_{beh} \cup t_{res} \vdash accept(team\ leader, location, number\ of\ team\ members). \quad \text{(A.4)}$$

The object agent refuses the task within the team allocation request only if:

$$t_{beh} \cup t_{res} \vdash \neg accept(team\ leader, location, number\ of\ team\ members). \quad \text{(A.5)}$$

B

Technical Description of Monitoring Process

In this Appendix we want to give the reader a clear, technical explanation of the monitoring process in collaborative environment.

Provided that we have Θ as set of all the agents in the community [1], meta-agent A^m, a set of all the object agents $\mu(A^m)$, an alliance members of the object agent A_i - $\mu(A_i)$ [1] we have the task τ as a triple $\langle loc(\tau), \varepsilon(\tau), tl(\tau)\rangle$:

- the task location $loc(\tau)$,
- the team leader $tl(\tau)$ and
- the team members $\varepsilon(\tau)$ defined in a such way [1]:

$$\varepsilon(\tau) \subseteq \mu(A_i) \cup tl(\tau). \tag{B.1}$$

The team allocation request from the team leader to the team member A_i can be either accepted or refused. If the request is accepted then the event belief formula is generated (the object agent has accepted the task within the team allocation request):

$$accept(A_i, tl(\tau), loc(\tau), |\varepsilon(\tau)|), \tag{B.2}$$

else the complementary event belief formula is generated (the object agent has refused the task within the team allocation request):

$$\neg accept(A_i, tl(\tau), loc(\tau), |\varepsilon(\tau)|). \tag{B.3}$$

The following event belief formula is generated in both cases, the team allocation request has been either accepted or refused (the team leader accepts the team counting of $|\varepsilon(\tau)|$ members):

$$\exists X \exists Y accept(tl(\tau), X, Y, |\varepsilon(\tau)|). \tag{B.4}$$

We will focus now on communication acts within the contract net protocol before the team allocation process. The object agent A_0 proposes the team to the object agent A_1 via proposed services and the deliver times. We will

estimate proposed team due to the reasoning simulation. The result of the reasoning simulation process (searching algorithm) consists of several teams, which any could be proposed.

Let us define a possible proposed team $\varepsilon p(A_0, \tau)_i$ by the object agent A_0 to the object agent A_1 within the team proposal for the mission m . There is a set of all the possible proposed teams $\varepsilon s(A_0, m) = \{\varepsilon p(A_0, m)_i\}$, where we are not sure which one has been really proposed. The event belief formula can be generated (the object agent A_0 accepts the object agent A_1 as the team leader in gbb) if this condition is valid:

$$\forall \varepsilon p(A_0, m)_i \in \varepsilon s(A_0, m) : \varepsilon p(A_0, m)_i \neq \emptyset. \tag{B.5}$$

Next event belief formula can be generated (the object agent A_0 accepts the object agent A_1 as the team leader and the object agent A_0 would participate in $loc(m)$ with services in gbb) if this condition is valid:

$$A_0 \in \bigcap_i \varepsilon p(A_0, m)_i. \tag{B.6}$$

The event belief formula can be generated (the object agent A_0 either does not accepts the object agent A_1 as the team leader or the object agent A_0 would not participate in $loc(m)$ with services in gbb) if this condition is valid:

$$\forall \varepsilon p(A_0, m)_i \in \varepsilon s(A_0, m) : A_0 \notin \varepsilon p(A_0, m)_i. \tag{B.7}$$

The last event belief formula says that the object agent A_0 accepts the team counting the team members:

$$\min_i(|\varepsilon p(A_0, m)_i|). \tag{B.8}$$

We suppose that the meta-agent is present in the system before any planning phase of the system. The condition is satisfiable in collaborative environment. The object agents adopt their decision making algorithm with respect to unsuccessful attempts to form a team. We can use the reasoning simulation for generation of event belief formulae about previous task rejections, although the meta-agent has lost some planning actions.

For every possible proposed team we define the set of the object agents, which could participate with services but they have not been included - $c\varepsilon p(A_0, m)_i$. The same as we defined the set of all possible proposed teams $\varepsilon s(A_0, m)$ we define a set of all teams of not included object agents $c\varepsilon s(A_0, m) = \{c\varepsilon p(A_0, m)_i\}$, which could participate. Let us suppose now that there is only one proposed team and only one agent which could participate and it has not been included in the team:

$$|c\varepsilon s(A_0, m)| = 1 \wedge |c\varepsilon p(A_0, m)_1| = 1, c\varepsilon p(A_0, m)_1 = \{A_K\}. \tag{B.9}$$

The task with the parameters $\langle loc(m), \varepsilon p(A_0, m)_1 \cup \{A_K\}, tl(m)\rangle$ has been refused earlier or the object agent A_0 does not accept the number of the team

members equivalent to the team $\varepsilon p(A_0, m)_1 \cup \{A_K\}$ [1]. It is valid:

$$\begin{array}{l} \neg \bigwedge_{A_i \in \sigma} accept(A_i, A_0, loc(m), |\varepsilon p(A_0, m)_1 \cup \{A_K\}|) \\ \vee \\ \neg \exists X accept(A_0, X, loc(m), |\varepsilon p(A_0, m)_1 \cup \{A_K\}|), \\ \\ where\ \sigma = (\varepsilon p(A_0, m)_1 \cup \{A_K\}) \setminus \{A_0\}. \end{array} \tag{B.10}$$

Now, let us suppose that:

$$|c\varepsilon s(A_0, m)| = 1 \wedge |c\varepsilon p(A_0, m)_1| \geq 1. \tag{B.11}$$

For all the object agents, which could participate but have not been included in the team, it is valid previous equation:

$$\begin{array}{l} \bigwedge_{A_j \in c\varepsilon p(A_0,m)_1} \\ (\\ \quad \neg \bigwedge_{A_i \in \sigma_j} accept(A_i, A_0, loc(m), |\varepsilon p(A_0, m)_1 \cup \{A_j\}|) \\ \quad \vee \\ \quad \neg \exists X accept(A_0, X, loc(m), |\varepsilon p(A_0, m)_1 \cup \{A_j\}|) \\ ), \\ where\ \sigma_j = (\varepsilon p(A_0, m)_1 \cup \{A_j\}) \setminus \{A_0\}. \end{array} \tag{B.12}$$

Let us suppose now that:

$$|c\varepsilon s(A_0, m)| \geq 1 \wedge |c\varepsilon p(A_0, m)_1| \geq 1. \tag{B.13}$$

Now we are not sure which possible proposed team is valid then previous equation is joined with a conjunction over all the possible proposed team:

$$\begin{array}{l} \bigvee_{c\varepsilon p(A_0,m)_k \in c\varepsilon s(A_0,m)} \\ (\\ \quad \bigwedge_{A_j \in c\varepsilon p(A_0,m)_k} \\ \quad (\\ \quad \quad \neg \bigwedge_{A_i \in \sigma_{jk}} accept(A_i, A_0, loc(m), |\varepsilon p(A_0, m)_k \cup \{A_j\}|) \\ \quad \quad \vee \\ \quad \quad \neg \exists X accept(A_0, X, loc(m), |\varepsilon p(A_0, m)_k \cup \{A_j\}|) \\ \quad) \\ ), \\ where\ \sigma_{jk} = (\varepsilon p(A_0, m)_k \cup \{A_j\}) \setminus \{A_0\}. \end{array} \tag{B.14}$$

Above discussion is valid only if

$$\forall c\varepsilon p(A_0, m)_k \in c\varepsilon s(A_0, m) : c\varepsilon p(A_0, m)_k \neq \emptyset. \tag{B.15}$$

If the condition does not hold then tautology event belief formula could be generated.

[1] We used the same functions (loc, tl) as defined above.

References

1. Pěchouček, M., Mařík, V., Bárta, J.: A knowledge-based approach to coalition formation. IEEE Intelligent Systems **17** (2002) 17–25
2. Pěchouček, M., Štěpánková, O., Mařík, V., Bárta, J.: Abstract architecture for meta-reasoning in multi-agent systems. In Mařík, Muller, P., ed.: Multi-Agent Systems and Applications III. Number 2691 in LNAI. Springer-Verlag, Heidelberg (2003)
3. McGuire, J., Kuokka, D., Weber, J., Tenebaum, J., Gruber, T., Olsen, G.: Shade: Technology for knowledge-based collaborative engineering. Concurrent Engineering: Research and Applications **1(3)** (1993)
4. Shen, W., Norrie, D., Barthes, J.: Multi-Agent Systems for Concurrent Intelligent Design and Manufacturing. Taylor and Francis, London (2001)
5. Decker, K., Sycara, K., Williamson, M.: Middle agents for internet. In: Proceedings of International Joint Conference on Artificial Intelligence 97, Japan (1997)
6. Sycara, K., Lu, J., Klusch, M., Widoff, S.: Dynamic service matchmaking among agents in open information environments. ACM SIGMOID Record **28** (1999) 211–246
7. Pěchouček, M., Mařík, V., Bárta, J., Tožička, J., Štěpánková O., Železný F.: Monitoring and meta-reasoning in multi-agent systems. Technical Report deliverable d.1, Research contract to AFRL – Air Force Research Laboratory, contract no.: FA8655-02-M4056, The Gerstner Laboratory, Czech Technical University in Prague (2002)
8. Tambe, M.: Towards flexible teamwork. Journal of Artificial Intelligence Research **7** (1997) 83 – 124
9. Mařík, V., Pěchouček, M., Štěpánková, O.: Social knowledge in multi-agent systems. In Luck, M., et al., eds.: Multi-Agent Systems and Applications. Number 2086 in LNAI. Springer-Verlag, Heidelberg (2001)
10. Maes, P.: Computational reflection. Technical report 87-2, Free University of Brussels, AI Lab (1987)
11. Pěchouček, M., Štěpánková, O., Mikšovský, P.: Maintenance of discovery knowledge. In Zitkow, R.J., ed.: Priciples of Data Mining and Knowledge Discovery. Springer-Verlag, Heidelberg (1999) 476–483
12. Grosz, B.J., Kraus, S.: The Evolution of SharePlans. In: Foundations of Theories of Rational Agency. (1999) 227–262

13. Kaminka, G.A., Tambe, M.: Robust multi-agent teams via socially-attentive monitoring. Journal of Artificial Intelligence Research **12** (2000) 105–147
14. Cao, W., Bian, C.G., Hartvigsen, G.: Achieving efficient cooperation in a multi-agent system: The twin-base modelling. In Kandzia, P., Klusch, M., eds.: Cooperative Information Agents. Number 1202 in LNAI. Springer-Verlag, Heidelberg (1997)
15. Štěpánková, O., Mařík, V., Lažanský, J.: Improving Cooperative Behaviour in Multiagent Systems. In: Advanced IT Tools. Chapman and Hall London (1996) 293–302
16. Huber, M.J., Durfee, E.H.: On acting together: Without communication. In: Working Notes of the AAAI Spring Symposium on Representing Mental States and Mechanisms. (1995) 60–71
17. Jennings, N.: Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. Artificial Intelligence **75(2)** (1995) 195–240
18. Kaminka, G., Pynadath, D., Tambe, M.: Monitoring deployed agent teams. In: Proceedings of the fifth international conference on Autonomous agents, Montreal, Quebec, Canada (2001)
19. Pěchouček, M., Macurek, F., Tichý, P., Štěpánková, O., Mařík, O.: Meta-agent: A workflow mediator in multi-agent systems. In: Intelligent Workflow and Process Management: The New Frontier for AI in Business. Morgan Kaufmann Publishers (1999)
20. Chang, C., Lee, C.R.: Symbolic Logic and Mechanical Theorem Proving. Academic Press New York and London (1973)
21. Kalman, J.: Automated Reasoning with Otter. Rinton Press, Inc., Prinecton, New Jersey (2001)
22. Robinson, J.A.: Automatic deduction with hyper-resolution. Internat. J. Comput. Math. **1** (1965) 227–234
23. Luger, G. F., S.W.A.: Artificial Intelligence: Structures and Strategies for Complex Problem Solving. The Benjamin/Cummings Publishing Company,Inc. (1993)
24. Loveland, D.: Iria symp. automatic demonstration, springer-verlag, new york. In: Conference on Automated Deduction. (1970) 147–162
25. Wos, L., Carson, D., Robinson, G.R.: Efficiency and completeness of the set of support strategy in theorem proving. JACM (1965) 698–709
26. Loveland, D.: Automated Theorem Proving: A Logical Basis. North-Holland Publishing Company (1978)
27. Loveland, D.: A simplified format for the model elimination procedure. JACM (1969) 349–363
28. Astrachan, O.: Meteor: Exploring model elimination theorem proving. Journal of Automated Reasoning (1994) 283–296
29. Letz, R.and Schumann, J., Bayerl, Bibel, L.: Setheo: A high-performance theorem prover. Journal of Automated Reasoning (1992) 183–212
30. Baumgartner, P., Furbach, U.: Model elimination without contrapositives and its application to PTTP. Journal of Automated Reasoning **13** (1994) 339–359
31. Astrachan, O., Loveland, D.: The use of lemmas in the model elimination procedure. Journal of Automated Reasoning (1997) 117–141
32. Astrachan, O.L., Stickel, M.E.: Caching and lemmaizing in model elimination theorem provers. In: Conference on Automated Deduction. (1992) 224–238
33. Mitchell, T.: Generalization as search. Artificial Intelligence (1982) 203–226

34. Muggleton, S., Raedt, L.D.: Inductive Logic Programming: Theory and Method. Journal of Logic Programming **19-20** (1994) 629–679
35. Zelezny, F., Srinivasan, A., Page, D.: Lattice-search runtime distributions may be heavy-tailed. In Matwin, S., Sammut, C., eds.: Proceedings of the 12th International Conference on Inductive Logic Programming (ILP'02). Number 3-540-00567-6 in LNAI. Springer-Verlag, Heidelberg (2002)
36. Wrobel, S.: First order theory refinement. In Raedt, L.D., ed.: Advances in Inductive Logic Programming. IOS Press (1996) 14–33
37. Bárta, J., Pěchouček, M., Mařík, V.: Sufferterra humanitarian crisis scenario. Technical Report GL 141/01. 35 p. ISSN 1213-3000, CTU FEE, Department of Cybernetics, The Gerstner Laboratory (2001)