Reflective-Cognitive Architecture: From Abstract Concept to Self-Adapting Agent

Lukáš Foltýn[1], Jan Tožička[1], Milan Rollo[2], Michal Pěchouček[1], Pavel Jisl[1]
Gerstner Laboratory[1] and Center for Applied Cybernetics[2]
Department of Cybernetics, Czech Technical University
Technická 2, Prague, 166 27, Czech Republic
{lfoltyn|tozicka|rollo|pechouc|jisl}@labe.felk.cvut.cz

Abstract

Multi-agent systems penetrate into interesting domains of computing with limited resources, programme code reusability and automated code generation. This paper presents general framework of Reflective-Cognitive agent architecture which enables the agent to alter its own code in runtime according to the changes in the environment. We also present results of architecture implementation showing the plausibility of created prototype.

1. Introduction

Multi-agent system consists of a number of agents running on a variety of host computers or other devices. System structure may dynamically change, agents may migrate among the hosts with diverse computational resources and even the parameters of the environment the agents operate in are usually not constant during the system's whole lifecycle. Agents thus have to adapt to these changes to keep the system functional.

There are two ways how to deal with this problem: (i) either the developer may take into account possible changes of the environment during the design time or (ii) reflective architecture that allows agent to adapt automatically can be used. This paper presents the second case - general framework of Reflective-Cognitive agent architecture which enables the agent to alter its own code in runtime. This architecture allows creation of lightweight agents that are able to adapt event to the situations that were not known during the design time.

Classically, in the software engineering we distinguish two types of reflection: **structural** [1, 3] and **behavioral** [6] reflection. In our work we adopt the interpretation published in recent contribution [8]: using the *Structural Reflection* the system structure can be dynamically modified, while *Computational (Behavioral) Reflection* modifies the system semantics (behavior). Examples of the respective cases provided by the authors are data structure modification and algorithm modification.

Applying the above definitions to the multi-agent reflection, we use structural reflection to change agent's private data – its state, acquaintance models, etc. The behavioral reflection is used to change the agent's reasoning algorithms – a part of the agent that interprets its data. The structural reflection can change agent's behavior in only limited way as all possible patterns of behavior have to be already implemented in agent code. The behavioral reflection implements an introspective force that is able to change the behavior of the agent in the most general way – on the code level, where we can compose or generate new algorithms incorporate them as agent's future behavior.

The paper is organized as follows. In Section 2 we present definitions of reflection in multi-agent systems, motivation for using reflection and possible component approach to the reflection. In Section 3 we present proposed Reflective-Cognitive agent architecture. Section 4 is devoted to the specific task of the reflective process – the creation of new algorithm. In the Section 5 we present an illustrative experiment with RC agent prototype within complex scenario.

2 Reflection in Multi-Agent Systems

A multi-agent reflection can be viewed on a macro- and micro- levels. This is why we distinguish between three different *types of reflection* in multi-agent systems. (i) **Individual Reflection** - revision of agents' isolated behavior, that does not necessarily need to result from agents' mutual interaction. Individual reflection is an operation that works with agent's awareness of its own knowledge, resources, and computational capacity. (ii) **Mutual Reflection** - revision of agent's interaction with another agent. This kind of revision is based on agent's knowledge about the other agent, trust and reputation, knowledge about the other

agent's available resources, possibly opponent's (or collaborator) longer term motivations and intentions (all these kinds of knowledge are refereed to as *social knowledge* and are stored in agent's acquaintance models). (iii) **Collective Reflection** - revision of agents' collective interaction. This is the most complex kind of reflection. Here we assume revision of the collective behavior of the group of agents as a result from their complex interaction. Collective reflection can be achieved either by: (a) a **single reflective agent** (e.g. a meta-agent) that is busy with monitoring the community behavior and updating agents' behavior or (b) **emergently** by the collective of agents, each carrying out its specific cognitive/reflective reasoning. In the collective reflection the agents update not only their social knowledge bases and reasoning processes but also they make the attempts to revise other agents' acquaintance models and possibly reasoning (unlike in the case of mutual reflection).

2.1 Motivation of Reflection Deployment

We will concentrate on the notion of autonomous adaptation using the reflection process described above, as this notion is critical for future open ubiquitous (pervasive, ambient) ad-hoc systems. Once we deploy the diverse elements of these systems, they must be able to integrate themselves into the functional organism and to maintain themselves operational even in a long-term perspective. Autonomous adaptation to the changing environment is critical, as it will significantly increase the usability of ubiquitous systems by (i) extending the system lifespan by increasing collaboration efficiency, (ii) extending the average lifespan as system will remain operational even after significant environmental or device changes, (iii) limiting or minimizing the human maintenance operations, (iv) enabling an easy extension of system functionality using the collective reflection process or (v) facilitating the transfer of the knowledge from the existing system into the new one during the replacement phase.

While we have specifically addressed the embedded, highly distributed multi-agent systems in the overview above, most points apply to all types of multi-agent systems.

Our motivation is to create an architecture which is able to adapt to the limited computational resources and changes in the environment by means of runtime code alternation, to share and incorporate previously unknown code and to keep this architecture as lightweight as possible including simple code development and deployment.

2.2 Component Approach

One of the goals of the architecture is to introduce a new approach how to implement agent's algorithms. We have decided to shatter agent's algorithms into smaller parts we refer to as *components*. The code splitting of the algorithm is done with respect to component's functionality (and reuse) or with respect to the data flow in the whole algorithm. Each component is a self-contained java class. Properly chained components form an algorithm which we call a *plan*. To build the plan, Reflective Layer (see below) uses the components as *black boxes* described by their inputs, outputs and meta-data. Plans can be triggered by an external or internal event and they implement agent's reaction to the event.

Advantages of component approach are as follows. (i) **Programme Code Sharing** - the code which is shared across several algorithms can be concentrated in a single component. (ii) **Programme Code Assembly** - using components to assembly algorithm gives us a great portion of freedom when trying to adjust the algorithm to the current state of the environment. Building blocks in the form of components are easy to understand and to manipulate. (iii) **Programme Code Exchange** - smaller portions of the code in the form of components are easier to exchange among agents than complex algorithms. Exchange of components among the agents allows a great deal of flexibility and at the same time the communication links are less loaded. Each agent can decide which components to integrate and combine them with its local code-base. (iv) **Programme Code Alternation** - the introduction of a single new component can result in alternation of multiple algorithms so that they can reflect new situations in the environment. Altering the code on the level of components is far more efficient than on the level of the whole algorithms. This concept is close to Aspect Oriented Programming (AOP) [5]. (v) **Algorithm Efficiency Improvement** - using component metrics provided by the Cognitive Module (see below), we can detect bottleneck of the algorithm and initiate a negotiation to find more efficient substitution. Considering component to be a black box and swapping it with component of the same functionality, enables easy manipulation with the algorithm code.

There is one *drawback* of the component-based approach. **Component Running Costs** – the principal disadvantage of the component approach is the cost of processing related to the data exchange among components. In order to minimize these costs, we concentrate on efficiency improvement of this code.

3. Reflective-Cognitive Agent Architecture

In a layered structure of the Reflective-Cognitive (RC) agent, the classical agent architecture forms the lower half of the structure – the *Reasoning Layer*. On the top of this layer, we add the *Reflective-Cognitive (RC) Layer*. The whole structure is shown in Figure 1.

Reasoning Layer handles regular agent operation – it performs agent-specific actions (interaction with agent-

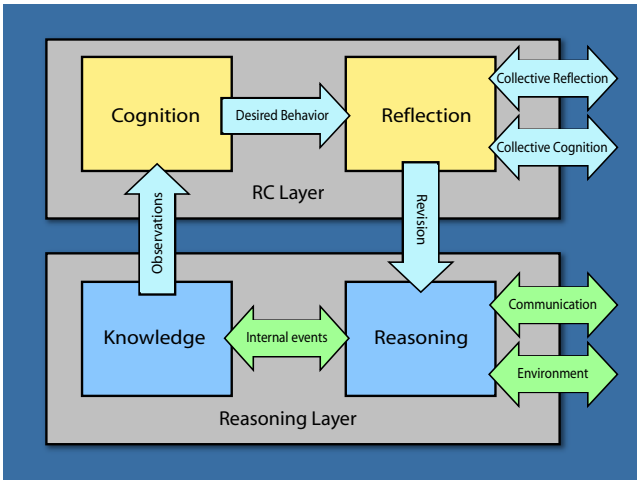

Figure 1. Reflective-Cognitive abstract architecture

specific technical resources, environment sensing), social interactions and finally planning and resource management.

To implement the reflective behavior, the **RC Layer** uses two main modules – *Cognitive Module*, that maintains the model of the agent in its environment and identifies the possible adjustments, and a *Reflective Module* that performs proposed modifications on the level of plans and components and delegates them to the Reasoning Layer.

3.1. Reasoning Layer

Reasoning Layer doesn't provide any reflective functionality. It stores and executes all agent's algorithms.

Algorithm instances (called sequences) are executed in a reaction to one of the following event types: (i) **environment event** which is invoked by the change of the surrounding environment, (ii) **internal event** which is invoked by the change of agent's internal state and (iii) **inter-agent communication** which is invoked by reception of a message from another agent.

The agent can also receive messages that it is not able to handle in an actual configuration. There can be two reasons for that: (i) RC Layer knows the appropriate processing algorithm, but decided not to load it into the Reasoning Layer, or (ii) appropriate algorithm is not present even in the RC Layer. In both cases, agent reacts to such messages by sending the *NOT-UNDERSTOOD* reply. Reflective process can later decide to incorporate the appropriate component or new algorithm (in the case (i)) or to query the other agents for appropriate components (case (ii)).

3.1.1 Knowledge Module

Components and plans need to share data which reflect agent's or environment's state. This is achieved using a structure called Knowledge Module. **Knowledge Module** is in our case implemented using *blackboard architecture*. Shared knowledge, stored and identified under *keys*, is accessible to all components in all algorithms. Knowledge alternation generates event which could be used for plan triggering.

3.1.2 Reasoning Module

The **Reasoning Module** is responsible for agent's functionality. Within the Reasoning Module all algorithms are maintained. It stores all used *plans*, all *components* used within these plans and all *sequences* (instances of running plans) – see Figure 2.

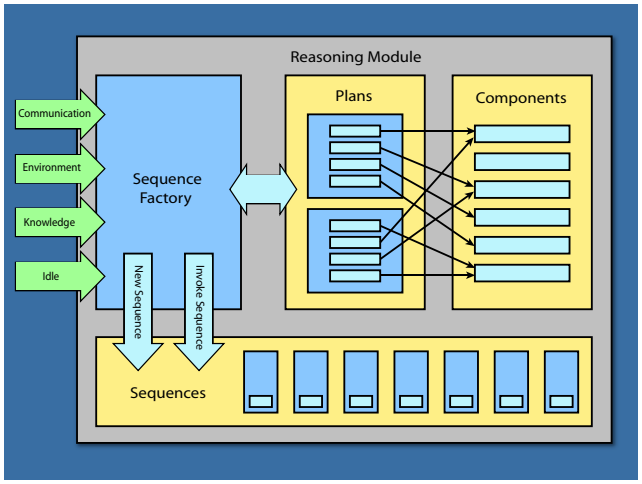

Figure 2. Reasoning Module – block scheme

After the event occurrence Sequence Factory selects appropriate plan from **Plans** storage and launches new **Sequence** object that manages execution of the plan.

From the technical point of view *component* is a piece of code, accompanied by *meta-description*, which could be externally executed and which typically produces well-defined outputs. Once triggered, the computational part of the component is executed. Usually code execution causes knowledge update or starts or continues in some message conversation.

While processing the plan, the sequence step by step invokes components, waits for their completion and provides them with a reference to their shared, plan-instance local data structure and the global blackboard. Therefore, local data structure modifications don't affect other plan instances. According to the output returned by the component after its completion, the next component from the plan is selected or the plan execution is finished. Each sequence runs in own thread.

3.2. Reflective-Cognitive Layer

The RC Layer is responsible for the reflective features of the architecture. It consists of two parts – *Cognitive Module*

and *Reflective Module*. The function of RC Layer is optimization of agent's behavior and its priority is low. Thus it is executed only in agent's idle time with relatively low probability.

3.2.1 Cognitive Module

The architecture of **Cognitive Module** is based on meta-agent abstract architecture [10]. It has two principal functions. It maintains *model* and it runs *meta-reasoning process* working on the model data.

The *model* contains the knowledge about the agent's environment, social neighborhood and the agent itself. The knowledge included in the model is less specific than the knowledge on the object level and it tends to represent historical experience rather than the current state only.

The goal of the cognition is to identify significant changes of the environment and to evaluate how effective/successfull are the actual algorithms using *meta-reasoning*. From the individual reflection's point of view various metrics can be used e.g. metrics based on the system load (the processor load or the memory consumed), agent's overall success (the ratio of won auctions), etc.

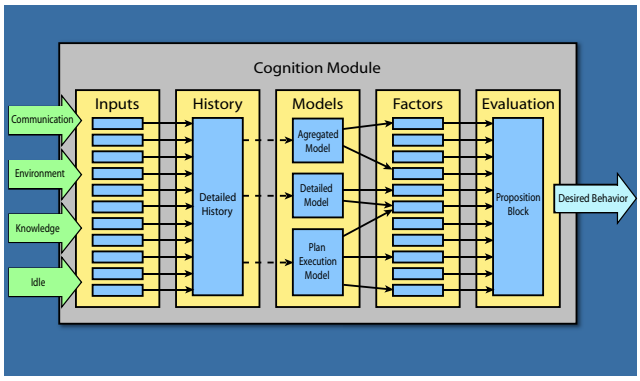

Figure 3. Cognitive Module – block scheme

Block Scheme of the Cognitive Module - The input data for the model in the cognition are gained by observation of the Reasoning Layer. To obtain fresh data from the Reasoning Layer, the cognition registers knowledge listener called **Inputs** within the blackboard. They update (asynchronously with the whole RC Layer) **History** component by data elements we call *Flags*. The information contained in the flag data part can be a very coarse digest of data from the knowledge. Each *Input* could be registered as a multiple event and knowledge listener and it could be very complex when transforming the knowledge. *Flags* are recorded in exact order as they have arrived thus causality of events is preserved. Once the RC process is triggered, **Models** are updated. With respect to data grouping we can distinguish two types of models - *Detailed* and *Aggregated*. In the *detailed* model we explicitly care about the order of events as they appeared. This gives us possibility to analyze causality of events in the system. In *aggregated* model we care about how many times certain event appeared between two dividing flags, thus providing us with pre-sorted data for statistical analysis. *Plan Execution Model* is special model which gathers data about plans' execution. This is used for plan retirement analysis. All *Models* are refreshed after the RC process was triggered. The models in the cognition contain information which is used by meta-reasoning to judge how good the overall behavior of the agent is. To do so, data from models is mapped to a set of real value attributes - **Factors** - which reflect various aspects of the environment including agent's behavior (e.g. successful delivery ratio or the current system load). For each component, a weight vector for cognition Factors as a part of meta-description is given. The weight value tells how strongly the component is affected by the factor. When an overall suitability of the plan is computed in **Evaluation** - *Proposition Block*, the weight vector for each algorithm component is multiplied by current value of factor. When certain algorithm gives low score of overall suitability, it is marked for reflection as a candidate for redesigning.

3.2.2 Reflective Module

Reflective Module has multiple functions. Its block structure is shown in Figure 4.

Reflective Action Identification block is responsible for handling of the behavior improvement specification received from the Cognitive Module (such as sudden system load drop notification) and execution of the appropriate block that can handle this specification. In general these blocks can be divided into three categories: (i) incorporation of default plans, (ii) replanning of actual plans and (iii) component generation. When the reflective process is invoked for the first time, *default plans* are loaded. They enable agent to operate from the beginning of its life-cycle (as all agent's algorithms are in the form of plans). Efficiency of default algorithms can be later improved by the reflective process.

If some plan is recognized by the Cognitive Module as not suitable for some reason, the request for *programme reconfiguration* is sent to the Reflective Module. Plan reconfiguration can be achieved either by replacement of one of the components within the plan or by modification of the whole plan. Besides planning sequences of appropriate components the Reflective Module can also be used for creation of new components – *code generation*, based on the observed events in the environment (see separate Chapter 4). Altered and new plans are passed to the Sequence Factory in the Reasoning Layer.

Plan Composition Approaches - When a Reflective Layer composes a new algorithm as a specific *event occurrence* handling (event-driven behavior), the forward plan-

ning algorithm will be used. If agent wants to reach a *new goal* (goal-driven behavior), backward planning algorithm is used. The planner searches the space of possible plans and selects the best one using the given utility function. Components within the plan are chained according to the rules stated in their meta-description.

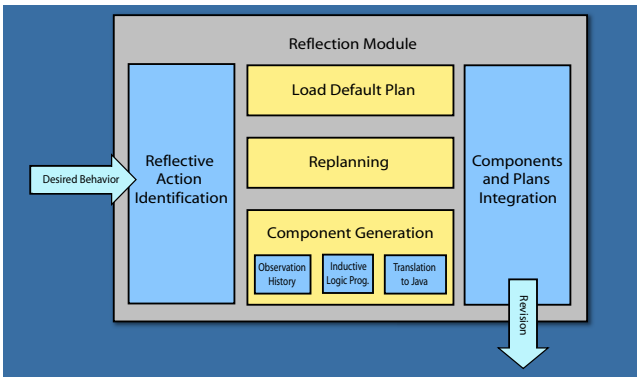

Figure 4. Reflective Module – block scheme

Algorithm Deactivation - in case of low rate or utility of a component or a plan, the cognition can suggest the reflection to remove it from all plans or to remove the whole plan. Component can be removed from the Reasoning Layer once the last running instance is completed. Note that the component code remains in the Reflective Layer and can be later incorporated in the Reasoning Layer again.

4. New Component Creation

The replanning can change the behavior of an RC agent by combining different components and thus adapt it to the dynamic environment. This approach covers a lot of different situations and can lead to the plans that have not been considered by system designer. Nevertheless, in some situations the agent does not know any component that could help it to solve some problem. In these situations, the agent can create new component. This process is invoked during the reflective process and can be very time consuming.

In our domain, we have implemented a component creation based on machine learning method ILP. Learned model is then translated into Java language and encapsulated into a component. In this prototype, we know what are the necessary inputs for this component, what type of knowledge this component produces and how to place this component into the created plan. All of these parameters are hardcoded into component creation module. In the future, we plan that component creation module should find out these properties autonomously. This will also allow the agent to create different types of components using the same module.

4.1. ILP

The subclass of machine learning methods known as *inductive logic programming* has been described e.g. in [7]. Briefly, ILP is a machine learning technique that can create a theory by generalizing given specific positive and negative examples. Created theory is common Prolog program, i.e. list of Horn rules.

Our principal finding is that it is feasible to implement the learning algorithm given the interpreted-Prolog (*tuProlog* interpreter [4] in our case) circumstance, resulting in the induction run-times peaking at tens of seconds on state-of-the-art hardware. An important question is however risen as to the scalability of this approach for future extensions of the problem setting (domain background knowledge, training data volumes, etc).

4.2. Prolog-to-Java Translation

As described above, risk evaluating rules created by ILP follow the syntax of Prolog programming language. In order to use these rules during the agent run without the necessity to run Prolog interpreter, we need to translate them into the Java language. Naturally, the Prolog rules are translated into Java *IF-THEN* statements. Once we have translated the rules into Java language it is necessary to encapsulate created code into a component. Newly created component is then incorporated into the agent's Plan knowledge base.

5. Experiments and Results

In this section we will demonstrate the plausibility of proposed architecture of reflective-cognitive agent. Presented experiments are just a small subpart the whole RC project and are intentionally selected to be simple and illustrative.

5.1. Case Description

The whole RC architecture including component creation has been implemented within ACROSS scenario [9]. In this scenario we solve a logistics problem in a competitive environment with self-interested agents. Java island is populated by location agents (cities with population) and transporter agents. Goods are created and consumed by locations and the surpluses are subject of trade among locations and delivered by transporters. Furthermore there are also several bandit agents that holdup loaded transporters. These bandits have their private restrictions where they will or will not rob the transporters they meet. Unlike common agent, thanks to its RC architecture, the transporter agent can create new component, that will predict the risk of bandit action (based on stored journeys history and bandit actions). Incorporating this component into its plans agent can reduce the risk of being robbed.

5.2. Experiment Setup

For our experiment we have run 18 cities spread out on the island and 6 bandits holding up the cargo transported by 10 transporter agents. Both the bandits and the transporters have been implemented using RC architecture. In addition, the transporters have been allowed to use ILP in order to create new component predicting the risk of hold-ups. Bandit have used a deterministic rule that specifies where the shall hold up the cargo.

The simulation has started without bandits. It has run 10 simulation days in order to reach some stable state. After that we have started the bandits and let the system stabilize in next 40 simulation days. At simulation day 50 we have allowed agents to use ILP.

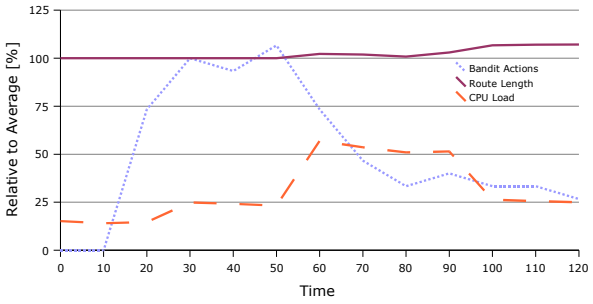

Figure 5. Course of the experiment when introducing adversarial actions in the system (day 10) and turning ILP on (day 50).

Figure 5 shows the trade-off between the number of adversarial actions and length of planned routes. Agent uses component creation mechanism to discover dangerous routes where adversarial agents are holding-up the transport. Observations sent to the ILP system are based on agent's own experience with behavior of the bandit agents. As soon as ILP creates a component for avoiding risky roads and the reflective process incorporates it into agent's plan, the number of adversarial actions significantly decreases. Expected consequence is that the average length of planned routes increases by several percents. These data are measured relatively to the *normal state* - no ILP running.

6. Conclusion

In this paper we have described the concept of agent reflection and cognition in distributed computational environment. The agents with RC architecture well adapt to the changes in surrounding environment and they better operate even in highly competitive or adversarial environments.

The *cognition*, agents' ability to understand the environment and its own reasoning process, is designed in a modular way so it can be easily extended to cover new properties of the system. Some parts of the cognition are domain independent (models, inputs for system load measurement, etc.) and some are domain dependent. The modular architecture of agent's *reasoning* layer can be easily updated or extended with new functionalities. The components (algorithm pieces) are easy to develop and deploy, although one must be careful with thread-related issues.

A series of experiments in a simulated logistic scenario have been undertaken (one of them has been presented in this contribution). These experiments lead us towards an assumptions that our RC architecture is well designed, lightweight and able to run on embedded devices. Integration on such devices would need to be carried out in order to provide proof-of-concept validation.

7. Acknowledgement

We acknowledge the support of the presented research by Army Research Laboratory project N62558-03-0819.

References

[1] S. Chiba. Load-time structural reflection in Java. In *proceedings of ECOOP 2000, LNCS*, volume 1850, pages 313–336. Springer-Verlag, 2000.

[2] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *proceedings of 2nd International Conference on Generative Programming and Component Engineering (GPCE '03), LNCS*, volume 2830, pages 364–376. Springer-Verlag, 2003.

[3] P. Cointe. A tutorial introduction to metaclass architecture as provides by class oriented languages. In *FGCS*, pages 592–608, 1988.

[4] E. Denti, A. Omicini, and A. Ricci. tuProlog: A light-weight prolog for internet applications and infrastructures. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages*. Springer, 2001.

[5] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[6] J. Malenfant, M. Jacques, and F. N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of the International Conference Reflection'96*, pages 1–20, San Francisco, CA, USA, 1996.

[7] S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[8] F. Ortin and J. M. Cueva. Non-restrictive computational reflection. *Comput. Stand. Interfaces*, 25(3):241–251, 2003.

[9] D. Šišlák, M. Rehák, M. Pěchouček, M. Rollo, and D. Pavlíček. $\mathcal{A}$-**globe**: Agent development platform with inaccessibility and mobility support. In R. Unland, M. Klusch, and M. Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 21–46, Berlin, 2005. Birkhauser Verlag.

[10] J. Tožička, J. Bárta, and M. Pěchouček. Meta-reasoning for agents' private knowledge detection. In M. Klusch, S. Ossowski, A. Omicini, and H. Laamanen, editors, *Cooperative Information Agent VII – Lecture Notes in Computer Science, LNAI 2782*, Springer-Verlag, Heidelberg, 2003.