A Flexible BDI Architecture Supporting Extensibility

Alexander Pokahr, Lars Braubach, Winfried Lamersdorf
Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
{pokahr | braubach | lamersd}@informatik.uni-hamburg.de

Abstract

The BDI agent model comprises a simple but efficient folk psychological framework of mentalistic notions usable for modeling rational agent behaviour. Nevertheless, despite its usefulness it is also a popular subject for extensions that try to improve the model in certain uncovered aspects such as emotions or norms. On the architectural level the BDI model is typically represented by an abstract BDI interpreter, which implements the fixed BDI reasoning cycle. In this paper it is argued that a fixed cycle has certain inherent drawbacks and that a transition towards a flexible agenda approach based on BDI meta-actions leads to a design open for extensions in many respects because new meta-actions can be easily integrated into the architecture on demand. To prove the validity of the approach, it is shown how the extensibility can be exploited to integrate concrete new aspects of increasing complexity into the model. They range from a simple mechanism for updating beliefs to a complex goal deliberation strategy and demand only slight modifications at well-defined extension points of the architecture. The new architecture as well as the presented extensions have been realized within the open source Jadex BDI reasoning engine.

1. Introduction

The Belief-Desire-Intention (BDI) model was conceived by Bratman as a theory of human practical reasoning [2]. Its success is based on its simplicity reducing the explanation framework for complex human behaviour to the *motivational stance* [6]. In this model, causes for actions are only related to desires ignoring other facets of cognition such as emotions. Another strength of the BDI model is the consistent usage of folk psychological notions similar to the way people communicate about human behaviour [17].

Even though BDI systems are used with considerable success in practice [13, 15] the BDI mechanism responsible for agent behaviour is simplistic in nature and hence cannot address many generic aspects of human behaviour and reasoning [17]. For this reason several different directions exist trying to extend the original model in various aspects. Thereby, the approaches can be coarsely subdivided into two subfields on the one hand aiming at improving the internal reasoning process and on the other hand extending BDI in the context of the agent's social environment. One direction of the former concerns the integration of emotions and BDI such as the TABASCO architecture [22]. Other approaches consider BDI and learning mechanisms [9] or the enhancement of goal representation and processing within the BDI architecture [4]. The second subfield treats teamwork issues addressed by architectures such as SimpleTeams [10] and the impact of sociological concepts like norms and obligations, e.g. within B-DOING [7].

To be implemented, all aforementioned kinds of approaches need to customize the BDI architecture to a greater or lesser extent. Therefore, extensibility of the BDI model is a crucial factor and should be reflected by a BDI architecture as well. The most prominent BDI architecture is represented by the abstract BDI interpreter (cf. Section 2.1), which defines a fixed control loop for the execution of agent behaviour. This approach has two main drawbacks:

First, the concrete layout of the cycle within the interpreter determines to a certain degree the agent's nature, e.g. it decides if an agent is cautious and reconsiders its choices often or if it is rather blindly-committed to its decisions. E.g. in [26] Wooldridge presents different versions of BDI interpreter cycles for blindly committed, single-minded and open-minded agent types and in [14] Kinny et al. show that the degree of the environmental dynamics influences the effectiveness of the different agent types. Hence, the definition of a *general applicable* interpreter cycle is extraordinarily difficult if not impossible, cf. Dastani et al. who say: "We assume that there is no unique (rational or universal) deliberation process and that the deliberation process can be specified in various ways" [5].

Second, a predefined interpreter cycle does not offer any apparent extension points for the integration of additional reasoning facilities, making it even more troublesome if several extensions need to be done. The BDI cycle prescribes that the reasoning process always is executed in a step by step manner of handling events, executing plan actions and finally updating mental structures. In this paper it is claimed that such a fixed cycle is very restrictive and renders extensions of the model hard to realize. Therefore, a more flexible way of mapping the BDI model to an architecture is proposed.

The remainder of this paper is structured as follows: In Section 2 an enhanced BDI architecture designed for extensibility is presented. To show the benefit of this flexibility some extensions of the BDI architecture are presented in Section 3. Section 4 describes how the basic interpreter as well as the proposed extensions have been realized in the open source Jadex BDI reasoning engine. The paper concludes with a summary and an outlook on future work.

2. A Flexible BDI Interpreter

Foundation for the new architecture is the original abstract BDI interpreter, what means that all functionalities from the classical approach are also incorporated into the new design. Therefore, the new approach is fully backwards compatible and does not change the way BDI agents can be programmed but only the way they are internally executed. In addition, it is construed to be very extensible allowing new facets being integrated into the architecture.

2.1. Original BDI Interpreter

Algorithm 1 Original BDI-interpreter, taken from [20]

```
01 initialize-state();
02 repeat
03     options := option-generator(event-queue);
04     selected-options := deliberate(options);
05     update-intentions(selected-options);
06     execute();
07     get-new-external-events();
08     drop-successful-attitudes();
09     drop-impossible-attitudes();
10 end repeat
```

The BDI theory of Rao and Georgeff [20] defines beliefs, desires, and intentions as mental attitudes represented as possible world states. The intentions of an agent are subsets of the beliefs and desires, i.e., an agent acts towards some of the world states it desires to be true and believes to be possible. To be computationally tractable Rao and Georgeff also proposed several simplifications to the theory, the most important one being that only beliefs are represented explicitly. Desires are reduced to events that are handled by predefined plan templates, and intentions are represented implicitly by the runtime stack of executed plans.

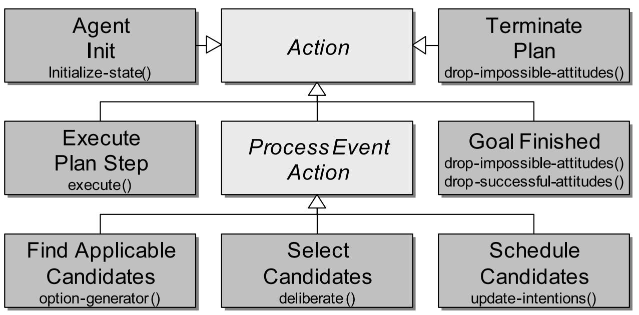

Figure 1. Identified meta-actions

The resulting abstract interpreter loop (see Alg. 1) captures the essence of early PRS systems [12] and is still the foundation for most current BDI-systems like JACK [11] and Jason [1]. Main task of the interpreter is to find and execute plans matching the given events and goals (lines 3-4). Afterwards, selected plans are executed (lines 5-6) and affected attitudes are updated accordingly (lines 8-9).

2.2. From Steps to Meta-Actions

In general, every agent architecture can be specified in terms of the agent state and the allowed state transitions. Given that an agent state is usually composed of cleanly separated elements such as beliefs, goals, and plans, the agent state can be easily extended by adding additional components (e.g. obligations) or by augmenting existing components (e.g. introducing context conditions for goals). The difficult part when extending the BDI (or some other) agent architecture, is to respect this new state information in the allowed state transitions. This might imply changing the architecture in one or more of the following ways: Introducing new state transitions, restricting/removing existing state transitions, or extending existing state transitions to also cope with newly introduced information.

From a software-engineering point of view it is desirable to minimize changes to existing transitions, when extending the architecture. On the one hand, because the behaviour of agents following the original architecture should remain the same (backwards compatibility). On the other hand, when realizing the extended architecture in an existing agent framework, only local changes have to be made to the code. The question is then: Which set of state transitions should be used to represent a flexible BDI architecture?

We claim that for an extensible agent architecture, software-engineering principles such as high cohesion and low coupling should be respected when defining a set of state transitions. The steps of the original interpreter already identify more or less separate functionality, and can be used as a starting point. The basic idea of the ar-

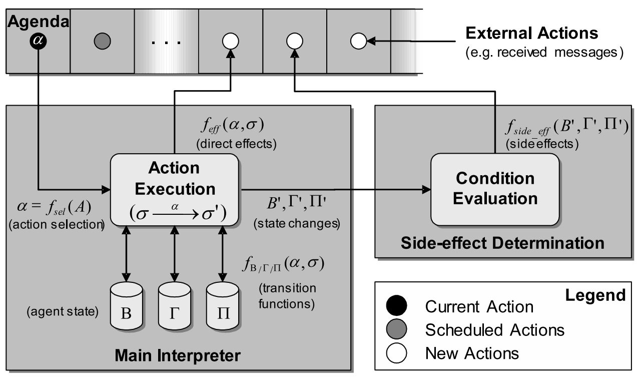

Figure 2. Interpreter architecture

chitecture is to break up the original abstract BDI interpreter cycle into a small set of self-contained meta-actions, which are invoked as needed, rather than being executed in a fixed sequence. Instead of generic steps operating on global data structures (like the execute() operation from line 6 in Alg. 1) these actions are instantiated on demand and include the exact elements on which to operate (e.g. the intention or plan step to execute). Fig. 1 shows the identified actions (dark rectangles) and introduces abstract actions (light rectangles) as well as inheritance relationships (arrows) to group similar actions. The names of the corresponding original interpreter steps are given below the action names.

2.3. Interpreter Architecture

The set of meta-actions forms the basis of the new interpreter architecture. Abandoning the view that all actions are executed as steps of a fixed interpreter cycle, the question arises how can be decided which action to execute next, and also, when should new actions be instantiated.

The basic mode of operation of the proposed interpreter is depicted in Fig. 2 (left hand side). The interpreter is based on a data structure called *Agenda* where all actions to be processed are collected. The interpreter continuously selects the next entry from the agenda and executes it, thereby changing the internal *agent state*. The execution of an action may lead to the creation of new actions (*direct effects*), which are then inserted into the agenda. In addition, certain occurrences may render the execution of already scheduled actions obsolete, e.g. an execute plan step action for a meanwhile dropped goal should not be performed. Hence, a precondition can be assigned to an action ensuring that obsolete actions are removed from the agenda.

The operation of the interpreter is first described independently of the details of concrete meta-actions and therefore different to BDI interpreter descriptions such as [19], in which the operation is primarily described in terms of a given set of basic actions. This paper does not intend to provide a complete operational semantics of the proposed architecture, but only highlights the important aspects. An agent state $\sigma \in \Sigma$ is defined as a tuple $\langle B, \Gamma, \Pi, A \rangle$, where B is a set of beliefs, Γ is a set of adopted goals, Π is a set of plans, and A is a set of scheduled agenda actions $\{\alpha, \alpha', \ldots\}$. An agenda action is a tuple $\langle \tau, \varphi_1, \varphi_2, \ldots \rangle$ with τ, an action type (e.g. *ProcessEvent* or *ExecutePlanStep*), and $\varphi_1, \varphi_2, \ldots$, parameters (type and number depending on the action type). For each action type, we introduce p_{pre}, f_B, f_Γ, f_Π, f_{eff}, which are characterising functions defined over $A \times \Sigma$. The precondition p_{pre} determines if the action is valid in the current context, the transition functions f_B, f_Γ, f_Π describe the changes in the beliefs, goals, and plans respectively, and the effect function f_{eff} determines a set of subsequent actions, which have to be added to the agenda. Note that it is not possible for an action to directly remove other agenda actions.

As result of the state transition caused by applying an agenda action α (written as $\sigma \xrightarrow{\alpha} \sigma'$ with $\sigma = \langle B, \Gamma, \Pi, A \rangle$, $\alpha \in A$ and $\sigma' = \langle B', \Gamma', \Pi', A' \rangle$), we get

$$B' = f_B(\alpha, \sigma),\ \Gamma' = f_\Gamma(\alpha, \sigma),\ \Pi' = f_\Pi(\alpha, \sigma),$$

$$A' = A \setminus \{\alpha\} \cup f_{eff}(\alpha, \sigma).$$

Note that actions are only performed when the precondition p_{pre} is valid. When the precondition does not hold, the action is dropped (B, Γ, Π stay unchanged and $A' = A \setminus \{\alpha\}$).

To allow flexible extension of the architecture, the creation of new agenda actions should not only depend on the direct effects $f_{eff}(\alpha, \sigma)$ of existing actions. E.g. to introduce creation conditions for goals, we do not want to change all actions that might possibly change the agent's beliefs. We therefore introduce the notion of side-effects, which are agenda actions, created due to changes in the agent state (see Fig. 2, right hand side). More formally, we define the side-effect function, which operates only on the changed beliefs, goals, and plans $f_{side\_eff}(B', \Gamma', \Pi')$ and determines additional actions to be added to the agenda. This leads to:

$$A' = A \setminus \{\alpha\} \cup f_{eff}(\alpha, \sigma) \cup f_{side\_eff}(B', \Gamma', \Pi')$$

External sources may also add entries to the agenda, such as messages that have been received from other agents and need to be processed. The state transition of adding such an externally created action α' to the agenda is unrelated to the execution of other actions, and therefore does not alter the other components of the agent state (i.e. $B' = B$, $\Gamma' = \Gamma$, $\Pi' = \Pi$, and $A' = A \cup \{\alpha'\}$).

Finally, a selection function $f_{sel}(A)$ is introduced (see Fig. 2), which is used by the interpreter to determine the next action to execute. As all scheduled actions have to be performed sooner or later, this selection does not represent a choice, but just an ordering. Therefore no complex reasoning is required, and simple strategies such as first-come-first-served can be applied. The basic operational model presented above already represents the complete interpreter. The details of the BDI architecture are realized as a set of self-contained meta-actions as described next.

2.4. Definition of Basic Meta-Actions

To further clarify the interpreter operation, we show how the basic actions (plan selection and execution) of a BDI agent are defined. Plan selection is implemented in the actions *FindApplicableCandidates*, *SelectCandidates*, *ScheduleCandidates*. Plan execution is done by the *ExecutePlanStep* action (cf. Fig. 1). The plan set Π of the agent may contain plan templates pt as well as plan instances given by $\pi = \langle pt, \varepsilon, \mu \rangle$, where ε is an event to handle (or $\perp$ if currently none), and μ is a counter (metrics) specifying the next step of the plan.

The $\langle FindApplicableCandidates, \varepsilon \rangle$ action α_{fac} produces a list of applicable plans for a given event ε. The effect of this action (leaving the beliefs, goals, and plans unchanged) is:

$$f_{\mathit{eff}}(\alpha_{\mathit{fac}}, \sigma) = \{\langle SelectCandidates, \varepsilon, \Pi_{app} \rangle\}$$

Thereby $\Pi_{app} = f_{app}(\Pi, \varepsilon)$ are the applicable plans for the event ε derived from the current plan set Π. Of the plan instances, only those are considered which do not currently have an event to handle (i.e. $\langle pt, \perp, \mu \rangle$).

The $\langle SelectCandidates, \varepsilon, \Pi_{app} \rangle$ action α_{sc} subsequently selects one or more plan templates or plan instances from the list of applicable plans. The effect of this action is therefore:

$$f_{\mathit{eff}}(\alpha_{sc}, \sigma) = \{\langle ScheduleCandidates, \varepsilon, \Pi_{can} \rangle\}$$

with $\Pi_{can} = f_{sel}(\Pi_{app}, \varepsilon)$ calculated by a plan selection function. For selected plan templates pt, the function returns a new plan instance $\pi = \langle pt, \perp, 0 \rangle$.

The $\langle ScheduleCandidates, \varepsilon, \Pi_{can} \rangle$ action α_{schc} updates all selected plan instances $\pi \in \Pi_{can}$ to include the event to be handled, thereby adding newly created plan instances to the plan set. Additionally, for each selected plan instance an *ExecutePlanStep* action is added to the agenda:

$$f_{\Pi}(\alpha_{schc}, \sigma) = \Pi \setminus \Pi_{can} \cup \Pi_{sched}$$

$$f_{\mathit{eff}}(\alpha_{schc}, \sigma) = \{\langle ExecutePlanStep, \pi \rangle \,|\pi \in \Pi_{sched}\}$$

$$\text{with } \Pi_{sched} = \{\langle pt, \varepsilon, \mu \rangle \mid \langle pt, \perp, \mu \rangle \in \Pi_{can}\}$$

Executing a plan step might change any aspect of the agent, depending on the code contained in the plan. Without going into details we represent those plan-induced changes using the transition functions $f_{B\pi}$, $f_{\Gamma\pi}$, $f_{\Pi\pi}$ defined over $\Pi \times \Sigma$. The functions are applied to the current state (i.e. $B' = f_{B\pi}(\pi, \sigma)$ and $\Gamma' = f_{\Gamma\pi}(\pi, \sigma)$). Additionally, the plan step counter of the plan instance is incremented by 1, and a new *ExecutePlanStep* action is added:

$$f_{\Pi}(\alpha_{eps}, \sigma) = f_{\Pi\pi}(\pi, \sigma) \setminus \{\pi\} \cup \{\pi'\}$$

$$f_{\mathit{eff}}(\alpha_{eps}, \sigma) = \{\langle ExecutePlanStep, \pi' \rangle\}$$

$$\text{with } \pi = \langle pt, \varepsilon, \mu \rangle \text{ and } \pi' = \langle pt, \varepsilon, \mu + 1 \rangle$$

In case the processing of the event is finished (indicated by a plan by waiting for a different event to process), the transition is slightly different, because the event is removed, and the plan does not have to be rescheduled:

$$\pi' = \langle pt, \perp, \mu + 1 \rangle, f_{\mathit{eff}}(\alpha_{eps}, \sigma) = \emptyset$$

3. Example BDI Extensions

In this section it is shown, how the presented architecture can be used to introduce extensions. Three cases of increasing complexity will be outlined.

3.1. Updating beliefs

For automatically refreshing the value of a belief that is e.g. connected to some sensor a new action $\alpha_{ub} = \langle UpdateBelief, \beta, \triangle t \rangle$ is introduced. It consists of the *UpdateBelief* action type and parameters for the specification of the belief and the update interval. As precondition for this action it is required that the belief is defined within the agent $p_{pre\_ub} = p_{is\_defined}(\alpha_{ub}, \sigma)$, i.e. the belief will not be updated if it is currently unknown by the agent, e.g. when the belief is removed by some plan at runtime. The results of an executed action are that the belief value is updated

$$f_B(\alpha_{ub}, \sigma) = B \setminus \{\beta\} \cup \{\beta'\}, \text{ with } \beta' = f_{ub}(\beta)$$

and that a new update belief action is added to the agenda

$$f_{\mathit{eff}}(\alpha_{ub}, \sigma) = \{\alpha'_{ub}\}, \text{ with } \alpha'_{ub} = \langle UpdateBelief, \beta', \triangle t \rangle$$

Note that the interpreter selection strategy has to ensure that only due actions are selected for execution. The action does not affect the agent's goals and plans (i.e. $\Gamma' = \Gamma$,$\Pi' = \Pi$).

3.2. Representing and Handling goals

In the original abstract BDI architecture [20] and most current implementations [1, 11] goals are represented only in the transient form of goal events. This implicit representation in combination with a purely procedural realization of goals was criticized because it hinders the agent in reasoning about its goals as no declarative information such as the goal's achievement state is available [4, 23, 25]. Hence,

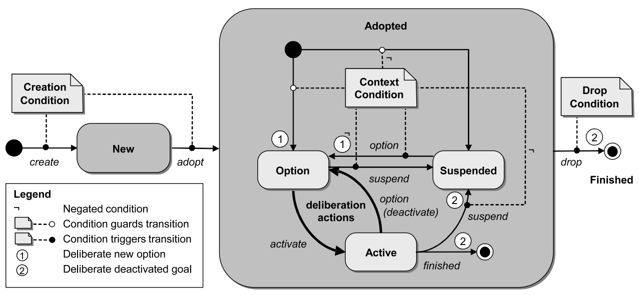

Figure 3. Goal lifecycle (adapted from [4])

an explicit representation of goals for BDI agent systems was conceived in [4]. In short, it consists of a generic goal lifecycle (cf. Fig. 3) for all supported goal types (perform, achieve, query, maintain) that exactly describes the states and transition relationships of goals at runtime. Adopted goals can be in either of the substates *Option*, *Active* or *Suspended*, whereby only active goals are currently pursued by the agent. The set of adopted goals is the union of the disjunctive sets of options, active and suspended goals $\Gamma = \Gamma_o \cup \Gamma_\alpha \cup \Gamma_\sigma$with $\Gamma_o \cap \Gamma_\alpha = \Gamma_o \cap \Gamma_\sigma = \Gamma_\alpha \cap \Gamma_\sigma = \emptyset$. Options and suspended goals represent inactive goals, where options are inactive, because the agent explicitly wants them to be, e.g. because an option conflicts with some active goal. In contrast, suspended goals currently must not be pursued, because their context is invalid. They will remain inactive until their context is valid again and they become options.

Additionally, some basic properties common to all goal types have been defined. Among those the most important ones are: A creation condition that defines when a new goal instance is created; a context condition that describes when a goal's execution should be suspended (to be resumed when the context is valid again); and a drop condition that defines when a goal instance is removed. Whenever such a condition triggers at runtime a corresponding goal meta-action is instantiated and inserted into the agenda, i.e. the state transitions are triggered as side-effects of the agenda execution from the formerly introduced function f_{side_eff}.

A goal $\gamma \in \Gamma$ is defined as a tuple $\langle gt, s \rangle$ with gt being the user defined goal template in which creation, context and drop condition among other things are specified and $s \in \{option, active, suspended\}$ being the actual state of the goal. For simplicity reasons other aspects of goals such as parameter values are not considered here.

For being able to handle a firing creation condition the new create goal action $\alpha_{cg} = \langle CreateGoal, gt \rangle$ is introduced. It is composed of the *CreateGoal* action type and the goal template gt as a parameter. Thereby, the goal template serves as model for goal creation, i.e. it contains relevant information for the created instance. The action is always applicable, i.e. the precondition is always true and it does only change the agent's goal state ($B' = B$, $\Pi' = \Pi$) by adding the new goal instance to the agent's set of goals.

$$f_\Gamma(\alpha_{cg}, \sigma) = \Gamma \cup \{\gamma\}\text{, with } \gamma = f_{create}(gt)$$

If the context condition of a goal triggers, two different cases with respect to its state have to be considered. For both cases the same action $\alpha_{sc} = \langle SwitchContext, \gamma, s' \rangle$ is used. It consists of the *SwitchContext* action type, the goal and the target state $s' \in \{option, suspended\}$, which is used to indicate whether the goal should be suspended (invalid context) or should be made an option (valid context). The action requires as precondition that the goal is not already in the target state $p_{pre\_sc} = (s \neq s')$ with $\gamma \in \Gamma$, $\gamma = \langle gt, s \rangle$. The application of the action leads to a changed goal state:

$$f_\Gamma(\alpha_{sc}, \sigma) = \Gamma \setminus \{\gamma\} \cup \{\gamma'\} \text{ with } \gamma' = \langle gt, s' \rangle$$

Finally, the dropping of an adopted goal is considered. The action $\alpha_{dg} = \langle DropGoal, \gamma \rangle$ is composed of the *DropGoal* action type and the goal to drop. Its precondition is always valid as only adopted goals $\gamma \in \Gamma$ are considered for dropping. As effect the goal is simply not contained in the set of adopted goals any longer:

$$f_\Gamma(\alpha_{sc}, \sigma) = \Gamma \setminus \{\gamma\}$$

3.3. Goal Deliberation

A well-known deficiency of the classical BDI model and architecture is the assumption that an agent can only posses consistent goals. It means that a BDI agent has no means for detecting whether some of its active goals interfere, leading to irrational behaviour. A typical example for such behaviour is a robot pursuing two movement goals with different target locations at the same time generating continuous to and fro. The consistency assumption is absolutely unrealistic as typical application scenarios involve a lot of different goals that interfere with each other either positively or negatively. Therefore in current BDI systems the agent programmer has the tedious and error-prone task to synchronize the agent's goals at the application level.

At this point goal deliberation strategies come into play. Such strategies are conceived to alleviate the goal arbitration task by shifting it from the application to the architecture level and by providing systematic means for specifying the interrelationships of goals and plans. This information is then used to detect interdependencies at runtime and preserve a consistent mental state. Currently, there is no consensus about how goal deliberation should be done and only a few approaches exist at all (e.g. [24]). In the following, it will be shortly sketched how a goal deliberation strategy called Easy Deliberation can be incorporated into the architecture. The strategy is based on ideas from goal modeling as can be found in the agent methodology Tropos [8] and the requirements engineering technique KAOS [16], which both propose directed contribution links between goals. Two main concepts are used to describe deliberation information within goal type declarations: *cardinalities* and *inhibition arcs*. Cardinalities can be used to constrain the maximum number of active goals of a specific type at runtime, whereas inhibition arcs are used to declare negative contribution relationships between two goals.

Generally two different situations can arise, in which deliberation becomes necessary (cf. Fig. 3): First, a goal can become an option either when a new goal is adopted or when the context of a suspended goal becomes valid again. In these cases the deliberation process needs to decide if the new option can be *activated* and additionally what the consequences of the activation are, i.e. which other active goals need to be *deactivated* to avoid having conflicting goals (*1: Deliberate new option*). Second, an active goal can become inactive when it gets suspended, finished or dropped. In this case, the deliberation has to determine which options have been possibly inhibited by the deactivated goal. For each of these options it needs to be checked if it can be reactivated (*2: Deliberate deactivated goal*).

The *Deliberate new option* meta-action $\alpha_{dno} = \langle DeliberateNewOption, \gamma \rangle$ is responsible to perform the activation of one option $\gamma \in \Gamma$, $\gamma = \langle gt, s \rangle$ if possible. The precondition has to verify that the goal currently is an option $p_{pre\_dno} = (s = \mathit{option})$. The effects of the action depend on the deliberation outcome given by the predicate $p_{activate}(\gamma)$. Only if it evaluates to true the activation is performed by making the option to an active goal and by making all inhibited active goals to options. The set of inhibited goals is calculated with the inhibition function $f_{inhibit}(\gamma, \gamma_x)$.

$$f_\Gamma(\alpha_{dno}, \sigma) = \Gamma \setminus \{\gamma\} \cup \{\langle gt, \mathit{active}\rangle\} \setminus \Gamma_{inh} \cup \Gamma_{opt}$$
$$\text{with } \Gamma_{inh} = \{\gamma_x \in \Gamma_\alpha \mid f_{inhibit}(\gamma, \gamma_x)\}$$
$$\text{and } \Gamma_{opt} = \{\langle gt, \mathit{option}\rangle \mid \langle gt, s\rangle \in \Gamma_{inh}\}$$

The *DeliberateDeactivatedGoal* meta-action $\alpha_{ddg} = \langle DeliberateDeactivatedGoal, \gamma \rangle$ is used to find out which current options could benefit from the deactivated goal $\gamma \in \Gamma, \gamma = \langle gt, s \rangle$. Precondition for this action being executed is that the goal still is not active $p_{pre\_ddg} = (s \neq \mathit{active})$, i.e. it was not reactivated in the meantime. As result of performing this action new *DeliberateNewOption* actions are produced for every option for which the deactivated goal was a *necessary condition* being not activated.

$$f_{\mathit{eff}}(\alpha_{ddg}, \sigma) = \{\langle DeliberateNewOption, \gamma_x\rangle \mid \gamma_x \in \Gamma_{inh}\}$$
$$\text{with } \Gamma_{inh} = \{\gamma_x \in \Gamma_o \mid f_{inhibit}(\gamma, \gamma_x)\}$$

More details of the Easy Deliberation strategy can be found in [18].

4. Interpreter Realization

The presented interpreter architecture has been realized in the Jadex BDI reasoning engine [3]. To follow the agenda-based execution model the Jadex reasoning engine has been extensively restructured. The core of the proposed architecture is realized within a component that is responsible for fetching and executing new meta-actions from the agenda whenever available. Components previously used in Jadex e.g. for plan selection and execution have been refactored to provide the basic set of required meta-actions.

All meta-actions are realized as separate Java classes implementing a common interface, which contains the precondition p_{pre} in form of an isValid() method. As the system is realized in an imperative language, the transition functions f_B, f_Γ, f_Π, f_{eff} are not represented separately, but merged into an execute() method, which performs the required state change. During action execution, the system automatically collects state changes which are used for the determination of side-effects. The side-effect function f_{side_eff} is implemented as a simple rule-base, where for each action the triggering condition is specified. When the condition holds after a state change, the corresponding action is added to the agenda. In case of arriving messages or internal timing events (that trigger e.g. belief updates or goal retries) the corresponding meta-actions are automatically created and added to the agenda.

Among others, the above presented BDI extensions have been integrated bit by bit. Currently, the system comprises 28 different meta-actions of varying complexity ranging from very simple implementations such as the *DropGoal* action to rather complex ones (e.g. the *ScheduleCandidate* action). The experiences gained from the development show that the different ways of integration can be used to adequately introduce new behaviour into the execution model. In addition, as long as extensions of the BDI model are not conceptually contradictory they can be used in conjunction. The belief update was integrated as an external action (triggered by a timing process), the explicit goal handling works with side-effects (triggered by goal conditions), and the goal deliberation uses direct effects (e.g. whenever a new goal is created the deliberation starts).

Due to the self-contained nature of the meta-actions and their very limited range of effects, a new extension in many cases does not require existing meta-actions to be changed. This is mainly owed to the fact that the meta-actions are usually belief, goal, or plan actions, which only affect a single component of the agent state and not a combination of beliefs, goals, and plans. Additionally, most actions are restricted to few instances (e.g. only change a single goal), leading to even more simple action specifications.

5. Conclusion

This paper tackles the question how the BDI architecture can be improved regarding the flexibility of extending the model by introducing new, or augmenting existing concepts. Extending the BDI model requires not only to extend the agent state representation, but also to adapt the interpreter to operate on this extended state. In this respect, deficiencies of the original BDI architecture are identified, in which state transitions are only implicitly represented as steps of an abstract interpreter.

In this paper a new architecture and system realization is presented, which is different from most cognitive agent architectures (such as [5, 20, 21]) in that it does not employ an interpreter cycle with a fixed set of steps. Instead, a flexible interpreter architecture is proposed, executing simple meta-actions from a dynamic agenda. The basic BDI meta-actions are derived from the original interpreter cycle leading to a backwards compatible execution model. This architecture supports flexibility in several ways. First it is easily possible to add new meta-actions. Moreover, due to the explicit representation in self-contained actions, changing the existing transitions is also simplified.

To verify the suitability of the approach, several extensions to the BDI model of different complexity have been presented. It is shown how an automatically triggered belief update mechanism, a framework for explicit handling of goals, and a goal deliberation strategy can be incorporated into the architecture. Both, the architecture as well as the presented extensions have been implemented within the open source Jadex BDI reasoning engine. As future work it is planned to exploit the extensibility of the architecture in further directions, e.g. one aspect we are especially interested in concerns teamwork in multi-agent systems.

References

[1] R. Bordini and J. Hübner. *Jason User Guide*, 2004. http://jason.sourceforge.net/.

[2] M. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.

[3] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A Short Overview. In *Net.ObjectDays 2004: AgentExpo*, 2004.

[4] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In *Proc. of Programming Multiagent Systems (ProMAS04)*, 2004.

[5] M. Dastani, F. Dignum, and J.-J. Meyer. Autonomy and Agent Deliberation. In *Proc. of the 1st International Workshop on Computatinal Autonomy (Autonomous 2003)*, 2003.

[6] D. Dennett. *The Intentional Stance*. Bradford, 1987.

[7] F. Dignum, D. Kinny, and E. Sonenberg. From Desires, Obligations and Norms to Goals. *Cognitive Science Quarterly*, 2(3-4):407–430, 2002.

[8] F. Giunchiglia, J. Mylopoulos, and A. Perini. The Tropos Software Development Methodology: Processes, Models and Diagrams. In *Proc. of Autonomous Agents and Multiagent Systems (AAMAS'02)*, 2002.

[9] A. Guerra-Hernández, A. El Fallah-Seghrouchni, and H. Soldano. Learning in BDI Multi-agent Systems. In J. Dix and J. Leite, editors, *Proc. of CLIMA IV*. Springer, 2004.

[10] A. Hodgson, R. Rönnquist, and P. Busetta. Specification of Coordinated Agent Behavior (The SimpleTeam Approach). In *Proc. of the Workshop on Team Behaviour and Plan Recognition at IJCAI-99*, 1999.

[11] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents-Summary of an Agent Infrastructure. In *Proc.of the 5th ACM Int.Conf. on Autonomous Agents*, 2001.

[12] F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 1996.

[13] F. Ingrand, M. Georgeff, and A. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert*, 7(6):34–44, 1992.

[14] D. Kinny and M. Georgeff. Commitment and effectiveness of situated agents. Technical Report 17, AAII, 1991.

[15] D. Kinny and R. Phillip. Building Composite Applications with Goal-Directed(TM) Agent Technology. *AgentLink News*, 16:6–8, December 2004.

[16] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. *SIGSOFT Softw. Eng. Notes*, 27(6):119–128, 2002.

[17] E. Norling. Folk Psychology for Human Modelling: Extending the BDI Paradigm. In *Proc. of Autonomous Agents and Multiagent Systems (AAMAS'04)*, 2004.

[18] A. Pokahr, L. Braubach, and W. Lamersdorf. A Goal Deliberation Strategy for BDI Agent Systems. In *Proc. of the 3rd German Multi-Agent Conference (MATES 2005)*, 2005.

[19] A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proc. of Modelling Autonomous Agents in a Multi-Agent World*, 1996.

[20] A. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proc. of the 1st Int. Conf. on MAS (ICMAS'95)*, 1995.

[21] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[22] A. Staller and P. Petta. Introducing Emotions into the Computational Study of Social Norms: A First Evaluation. *Artificial Societies and Social Simulation*, 4(1), 2001.

[23] J. Thangarajah, L. Padgham, and J. Harland. Representation and Reasoning for Goals in BDI Agents. In *Proc. of the 25th Australasian Computer Science Conf. (ACSC2002)*, 2002.

[24] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and Avoiding Interference Between Goals in Intelligent Agents. In *Proc. of the 18th Int. Joint Conf. on AI (IJCAI 2003)*, 2003.

[25] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & Procedural Goals in Intelligent Agent Systems. In *Proc. of the 8th Int. Conf. on Principles and Knowledge Rep. and Reasoning (KR-02)*, 2002.

[26] M. Wooldridge. *Reasoning about Rational Agents*. Intelligent Robots and Autonomous Agents. The MIT Press, Cambridge, Massachusetts, 2000.