Modular BDI Architecture

Peter Novák [*]
peter.novak@in.tu-clausthal.de

Jürgen Dix
dix@tu-clausthal.de

Department of Computer Science
Clausthal University of Technology
Julius-Albert-Str. 4, D-38678 Clausthal-Zellerfeld, Germany

ABSTRACT

One of the main challenges in agent-oriented programming is the design of *specialized programming languages* for single agent development. They should provide transparent interfaces to existing mainstream programming languages for easy integration with external code and legacy software. The underlying architecture of such programming languages has to be robust enough to support various approaches to knowledge representation and agent reasoning models.

In this paper we propose a modular BDI agent programming architecture, which is independent of the internal structure of its components and agent reasoning model. The connections between the components of such a BDI system are provided by interaction rules. Using this separation, we are able to draw a clear distinction between knowledge representation issues of a BDI agent system components and its dynamics.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*

General Terms

Languages, Theory

Keywords

agent and multiagent architectures; agent programming languages; frameworks, infrastructures and environments for agent systems; BDI architecture

[*]Primary author of this paper is a PhD. student.

1. INTRODUCTION

In [15] Rao and Georgeff introduced an abstract architecture for rational agents based on BDI logic together with a proposal for a practical implementation of this architecture in a software system. Their proposal introduced three data structures for *beliefs*, *goals* and *intentions*, relations and interactions of which are governed by a set of basic axioms of rationality [14].

Apart from a mass of theoretical work inspired by this abstract architecture a number of programming languages for development of rational agents have been implemented as well. State-of-the-art BDI agent programming frameworks implementing this abstract schema fall into two main groups. On the one hand, there are agent programming platforms like Jadex [12] and JACK [9, 17], in which agent programming support is integrated into an existing programming language by addition of a layer of specialized agent oriented programming constructs. The main advantage of this approach is to enable clear and transparent integration of agent system with external and legacy software systems. Additionally, these platforms provide access to a wide range of supporting programming libraries of the underlying programming language. As the main motivation for these systems is to exploit the strengths of mainstream programming frameworks, these approaches lack a strong formal semantics defined in terms of BDI logic. Therefore a formal investigation of agent system properties relies more or less only on the semantics of the underlying programming language.

On the other hand, standalone agent programming languages like AgentSpeak(L) [3, 13] and 3APL [5, 8] provide a clear semantics of the agent system. Most of these programming languages build on the use of logic based knowledge representation and support a declarative way of encoding the dynamics of the agent system. This allows deeper theoretical insights into the behaviour and the properties of systems built with these programming languages and allows the use of formal methods like verification, or model checking. Unfortunately, they usually provide only a basic interface to external software systems. Because of this lack of integration with mainstream programming languages and various knowledge representation techniques, they still remain more theoretical tools than widely applied software development environments.

Clearly, if agent oriented programming languages are going to gain ground in software engineering in general, besides providing a solid theoretical basis for studying properties of applications built with them, they also have to support traditional software development techniques to the largest possible extent. Therefore programming languages providing transparent integration with legacy systems and robust interfaces to external software systems are of high interest for agent oriented programming community [11].

In this paper, we address this issue and propose a modular BDI programming architecture to bridge this gap (Section 2). We describe syntax and semantics of an abstract programming language, together with an interpreter providing the dynamics of the system. The architecture consists of independent components for beliefs, desires, intentions and agent's capabilities, which are glued together by rules governing their interactions in terms of component queries and updates. This distinction provides a *separation of concerns* of knowledge representation issues for individual BDI components and the agent system dynamics encoding. It also allows the exploitation of a wide range of knowledge representation techniques, programming languages and libraries.

We do not focus specifically on development of rational agents in the traditional BDI sense. Rather we consider more general schemes, in which the rational agent reasoning model is only one of their possible specializations. We also show how various agent models can be implemented in it (Section 3) and discuss its relation to existing agent programming systems (Section 4). We conclude by pointing out directions for future work (Section 5).

2. MODULAR BDI ARCHITECTURE

The inspiration for our work was a particular aspect of the agent programming language 3APL [8], in which the interactions between BDI components are independent of the internal structure of agent's belief base. Essentially, only a query and update interfaces are important for interactions of a belief base with other components of a 3APL agent system.

Our main idea is to *generalize this independence of components* also for goals and intentions of BDI agent schema, thus allowing replacement of individual modules of an agent system as far as they provide a proper interface which integrates with the rest of the BDI system.

Our BDI agent consists of four basic components for *beliefs*, *desires*, *intentions* and *capabilities*, where the last one serves as a technical shortcut for encapsulating implementation of agent's capabilities w.r.t. its environment (i.e. agent's *sensors* and *effectors* influencing the environment). Each of these components provides an interface for *query* and *update*. Interactions between components of an agent are defined by *interaction rules* similar to usual production rules of the form: *"if query Q holds, then update U is performed"*. The dynamics of the whole system is provided by *selection* and *execution* of interaction rules by an interpreter. This is provided by an interpreter. By specialising the rule selection and imposing constraints on rule types allowed, we can implement various agent reasoning models (see Section 3).

Figure 1 depicts a schema of the proposed BDI agent system architecture. Replaceable modules of a BDI system (depicted as rectangles with round corners) provide query and update interfaces. Apart from using these basic operations, no other means of interaction between components of a BDI system is considered. When an interpreter executes an interaction rule, the query part of it (left side of a rule) is consulted, and when evaluated to true, the update part (right side) is executed. The result of such an execution is an update of a certain BDI component. In the subscript of

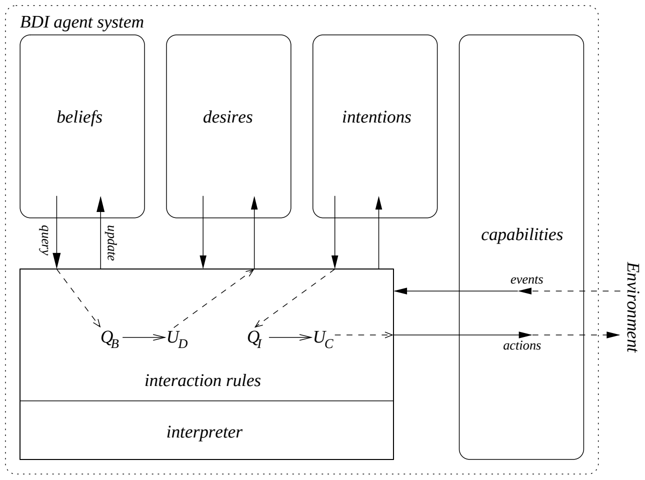

Figure 1: BDI Agent System

query and update expressions, the consulted component is indicated (depicted by dashed arrows). Agent's interaction with the outer environment is facilitated by its capabilities. Events are perceived by sensory capabilities, so that when the base of capabilities is queried, it provides information about state of the outer environment and events occurred in it. Based upon its mental state, an agent can act in its environment by updating its base of capabilities, what in turn results in performing an action by agent's effectors.

Note that we consider only the run-time representation of an agent. Design-time form, or syntax of an agent programming language associated with our framework, is not in the scope of this paper.

2.1 Syntax

In order to investigate the strengths of various approaches to knowledge representation (for implementation of a component of BDI system), we need an abstract programming language general enough to accommodate interfaces of wide range of programming languages from Prolog to C++, or LISP. Therefore we reduce the requirements on this language as much as possible and assume only the existence of generic domains and variables over them.

Definition 1. ***(BDI component languages)*** *Let $D_1, \ldots, D_n$ be a set of domains and $Var_{D_1}, \ldots, Var_{D_n}$ be mutually disjoint sets of corresponding domain variables. Let $\mathcal{L}_B$, $\mathcal{L}_D$, $\mathcal{L}_I$, $\mathcal{L}_C$ be languages possibly involving elements of $D_1, \ldots, D_n$ and $Var_{D_1}, \ldots, Var_{D_n}$.*

We call $\mathcal{L}_B$, $\mathcal{L}_D$, $\mathcal{L}_I$, $\mathcal{L}_C$ BDI component languages of beliefs, desires, intentions and capabilities with typical element belief, desire, intention and capability respectively. We also assume w.l.o.g., that BDI component languages are mutually disjoint.

Domains and domain variables over them are elements of BDI component languages which can be shared between theories in these languages. Typical example would be domain of natural numbers and set of variables standing for them. From this point on, we will denote formulas of BDI component languages by φ. Formulas which do not include variables are called *ground* and those which do are *non-ground.*

From formulas of BDI component languages, query and update expressions can be constructed.

DEFINITION 2. ***(query and update)*** *Let $\mathcal{L}$ be a BDI component language. Then query and update languages $\mathcal{L}_Q$ and $\mathcal{L}_U$ are defined as follows:*

- *if $\varphi \in \mathcal{L}$, then $Q(\varphi) \in \mathcal{L}_Q$ and $U(\varphi) \in \mathcal{L}_U$,*
- *if $\phi_1, \phi_2 \in \mathcal{L}_Q$ then $\phi_1 \wedge \phi_2 \in \mathcal{L}_Q$,*
- *if $\psi_1, \psi_2 \in \mathcal{L}_U$ then $\psi_1 \circ \psi_2 \in \mathcal{L}_U$.*

EXAMPLE 1. ***(running example)*** *Consider a simple crane agent which is able to take/put a container from/to a cart. The belief base of the agent is implemented using C++ and contains the following code chunk:*

```
class CCraneBelief {
public:
  bool isHolding(int & nContID) const;
  void tookCont(int nContID);
  bool isCartPrepared() const;
  void cartArrived(bool bValue);

  int m_nContId;
  bool m_bCartOK;
  ...
};
static CCraneBelief crane;
```

Querying and updating the belief base corresponds to execution of valid C++ expression invoking the interface of object crane. *Sample query and update formulas look as follows:*

$$Q(\text{crane.isHolding(2)}) \wedge Q(\text{crane.isCartPrepared()}) \quad (1)$$

$$U(\text{crane.tookCont(3)}) \quad (2)$$

As BDI component languages can be arbitrarily expressive, we consider only conjunctions of query formulas and sequences of updates. More complex binary connectives of queries and updates are left to abilities of particular BDI component language.

Intersection of any two BDI component languages is empty, therefore each query and update formula is uniquely associated with only one BDI component language. However, for clarity, we will indicate the corresponding BDI component language in the subscript of Q and U whenever necessary. From this point on, we will usually denote query formulas by ϕ and update formulas by ψ.

Interactions between components of a BDI agent system are defined by interaction rules as follows.

DEFINITION 3. ***(interaction rules)*** *Let $\mathcal{L}_Q$ and $\mathcal{L}_U$ be query and update languages, $\phi \in \mathcal{L}_Q$ and $\psi \in \mathcal{L}_U$. We say that a rule of the form*

$$\phi \longrightarrow \psi$$

is an interaction rule.

The informal semantics of interaction rules is that of production rules. If the query ϕ is true w.r.t. given theory from a BDI component language, then the update operation defined by update formula ψ can be performed on another theory.

2.2 Semantics

The formal semantics of our BDI system is, in tradition with other BDI agent programming languages such as AgentSpeak(L) and 3APL, provided in terms of a transition system. A transition system is a set of derivation rules for deriving transitions. A transition is a transformation of one system configuration into another and it corresponds to a single computation step. The configuration of a BDI agent is a record of its current mental state at a certain point of time. Components of a BDI agent are defined as theories in given BDI component languages. Transitions between them are performed via execution of interaction rules.

DEFINITION 4. ***(BDI agent configuration)*** *A BDI agent configuration is $(\beta, \delta, \iota, \kappa)$, a tuple of theories, where theory $\beta \subseteq \mathcal{L}_B$ is a belief base of an agent, $\delta \subseteq \mathcal{L}_D$ is the base of desires, $\iota \subseteq \mathcal{L}_I$ stands for the base of intentions of it and finally $\kappa \subseteq \mathcal{L}_C$ represents the base of capabilities.*

A BDI agent configuration is a run-time representation of an agent.

Bases of beliefs, desires, intentions and capabilities are structures storing information about corresponding aspect of agent's mental state: they provide interfaces for its manipulation. Formally, they are represented as theories in corresponding BDI component languages. From now on, we will formally speak about theories in languages of individual BDI components β, δ, ι and κ and on the technical level we will use term *base* for them.

EXAMPLE 2. ***(running example cont.)*** *A belief base of the crane agent from Example 1 at a certain point in time is a state of member variables of the* crane *object. If, let's say,* m_nContId=2 *and* m_bCartOK= true*, the agent believes that it is holding a container No. 2 and the cart is prepared to be loaded.*

In order to evaluate queries and perform updates on BDI components we define semantical operators *Query* and *Update*.

DEFINITION 5. ***(query and update operators)*** *Let $\mathcal{L}$ be a BDI component language and $\tau \subseteq \mathcal{L}$ be a theory in that language. $\mathcal{L}_G \subseteq \mathcal{L}$ is the set of all ground formulas from $\mathcal{L}$ and $\varphi, \psi \in \mathcal{L}_G$ formulas in it .*

A query operator $Query_{\mathcal{L}}$ associated with $\mathcal{L}$ is a mapping

$$Query_{\mathcal{L}} : \mathcal{L}_G \times 2^{\mathcal{L}} \to \{true, false\}; \langle \varphi, \tau \rangle \mapsto Query_{\mathcal{L}}(\varphi, \tau)$$

Similarly, corresponding update operator $Update_{\mathcal{L}}$ is a mapping

$$Update_{\mathcal{L}} : \mathcal{L}_G \times 2^{\mathcal{L}} \to 2^{\mathcal{L}}; \langle \psi, \tau \rangle \mapsto Update_{\mathcal{L}}(\psi, \tau)$$

In the case of a query, the result is truth value of φ w.r.t. τ. The result of an update operator application is a new theory $\tau' \subseteq \mathcal{L}$ which reflects an update ψ on τ.

Query and update operators $Query_{\mathcal{L}}$ and $Update_{\mathcal{L}}$ should be computable procedures evaluating formula $\varphi \in \mathcal{L}$ against theory $\tau \subseteq \mathcal{L}$. Especially in the field of logic programming and non-monotonic reasoning, there's a vast number of theories dealing with knowledge base updates (for overview see e.g. [10]). For many of them, there are even implementations of update operators. We can exploit capabilities and strengths of these approaches for implementation of various update operators in specializations of our architecture.

Example 3. ***(running example cont.)*** *Query and update operators for the belief base, as they were introduced in Example 1, simply evaluate the given C++ expression against current state of the belief base*[1]. *The query is true, when the return value of the executed expression is also* true *(i.e. not equal* 0*).*

By the query formula 1 from Example 1, we consult the belief base to find out whether our crane agent beliefs that it is holding container No. 2 and at the same time also believes that the cart, into which the container is to be put, is prepared for loading. After execution of the update formula 2, the crane agent should believe that it took the container No. 3, so now it holds it.

In order to define the semantics of interaction rules in terms of queries and updates of BDI components, we first need to define a semantics of these query and update formulas. We do this by relating formulas of the form $Q_{\mathcal{L}}(\varphi)$ and $U_{\mathcal{L}}(\psi)$ to applications of corresponding operators $Query_{\mathcal{L}}$ and $Update_{\mathcal{L}}$.

Definition 6. ***(semantics of ground query and update formulas)*** *Let $\mathcal{L}_1, \ldots, \mathcal{L}_4$ be a quadruple of BDI component languages and $\mathcal{L}_Q$ and $\mathcal{L}_U$ be query and update languages over union of these disjoint BDI component languages. Let also $(\tau_1, \tau_2, \tau_3, \tau_4)$ be a BDI agent configuration with $\tau_i \subseteq \mathcal{L}_i$ for $1 \leq i \leq 4$. Query operator is denoted by $\models$ and $\oplus$ denotes an update operator.*

The semantics of a ground query formula $\phi \in \mathcal{L}_Q$ is defined as follows

- *if $\phi = Q(\varphi)$ and $\varphi \in \mathcal{L}_i$ then $(\tau_1, \tau_2, \tau_3, \tau_4) \models \phi$ iff $Query_{\mathcal{L}_i}(\varphi, \tau_i) = true$,*
- *if $\phi = Q(\varphi_1) \wedge \cdots \wedge Q(\varphi_n)$ for $n > 1$, then $(\tau_1, \tau_2, \tau_3, \tau_4) \models \phi$ iff for each φ_i holds $(\tau_1, \tau_2, \tau_3, \tau_4) \models Q(\varphi_i)$ for $1 \leq i \leq n$.*

and for a ground update formula $\psi \in \mathcal{L}_U$

- *if $\psi = U(\varphi)$ and $\varphi \in \mathcal{L}_i$, then $(\tau_1, \tau_2, \tau_3, \tau_4) \oplus \psi = (\tau_1', \tau_2', \tau_3', \tau_4')$ iff $Update_{\mathcal{L}_i}(\tau_i, \varphi) = \tau_i'$ and $\forall j \neq i : \tau_j' = \tau_j$,*
- *if $\psi = U(\varphi_1) \circ \cdots \circ U(\varphi_n)$ for $n > 1$, then $(\tau_1, \tau_2, \tau_3, \tau_4) \oplus \psi = (\tau_1', \tau_2', \tau_3', \tau_4')$ iff there exists a chain of configurations $(\sigma_{1,1}, \sigma_{2,1}, \sigma_{3,1}, \sigma_{4,1}), \ldots, (\sigma_{1,n+1}, \sigma_{2,n+1}, \sigma_{3,n+1}, \sigma_{4,n+1})$, such that $(\tau_1, \tau_2, \tau_3, \tau_4) = (\sigma_{1,1}, \sigma_{2,1}, \sigma_{3,1}, \sigma_{4,1})$, $(\tau_1', \tau_2', \tau_3', \tau_4') = (\sigma_{1,n+1}, \sigma_{2,n+1}, \sigma_{3,n+1}, \sigma_{4,n+1})$ and for each i, $1 \leq i \leq n$, holds $(\sigma_{1,i+1}, \sigma_{2,i+1}, \sigma_{3,i+1}, \sigma_{4,i+1}) = (\sigma_{1,i}, \sigma_{2,i}, \sigma_{3,i}, \sigma_{4,i}) \oplus U(\varphi_i)$.*

Informally, an update formula $U(\varphi_1) \circ \cdots \circ U(\varphi_n)$ is evaluated from left to right, where each successive update is applied to the result of the previous one. Since different orders of updating a configuration can lead to different results, note that the operation of updates chaining $\circ$ is not commutative.

Example 4. ***(running example cont.)*** *Assume a BDI agent configuration with the current state of the belief base as it was described in Example 2 and semantics of Query and Update operators from Example 3. The query formula 1 from Example 1 is true and after execution of the update formula 2 the agent should believe that it is holding container 3. I.e. the result of this update operation is a configuration in which the belief base of an agent is modified so that* m_nContId=3.

For evaluation of a query formula against a BDI component theory by *Query* operator, the formula should be ground. Similarly for update formulas. However, non-ground formulas provide a more concise form of programming and therefore we have to deal with them. Transformation of non-ground formulas to ground ones is provided by means of *variable substitution*. Variable substitution is a mapping $\theta \subseteq \{[V/T] | V \in Var_{D_i} \wedge T \in D_i \wedge 0 \leq i \leq n\}$, where $D_1, \ldots D_n$ are domains and $Var_{D_1}, \ldots, Var_{D_n}$ sets of corresponding domain variables. By $\varphi\theta$ we denote a ground formula with all occurrences of variable $V \in Var_{D_k}$ replaced by an element $T \in D_k$, such that $[V/T] \in \theta$ for some $0 \leq k \leq n$.

Finally, for a query formula $\phi \in \mathcal{L}_Q$ we denote by $\phi\theta$ a formula $Q(\varphi\theta)$ for $\phi = Q(\varphi)$. In the case that $\phi = Q(\varphi_1) \wedge \cdots \wedge Q(\varphi_n)$, then $\phi\theta = Q(\varphi_1\theta) \wedge \cdots \wedge Q(\varphi_n\theta)$. Analogically for update formulas.

We say that variable substitution θ is ground w.r.t. formula ϕ, when $\phi\theta$ does not contain any variables.

At each time point of a system life cycle, queries of only a subset of all interaction rules hold. We say that an interaction rule $\phi \longrightarrow \psi$ is *applicable* to a BDI agent configuration $(\beta, \delta, \iota, \kappa)$ iff there exists a ground variable substitution θ, such that $(\beta, \delta, \iota, \kappa) \models \phi\theta$.

An agent system moves from one configuration to another exclusively by the application of some currently applicable interaction rule. All possible agent configurations are connected to each other via transitions from an associated transition system induced by interaction rules. More formally

Definition 7. ***(agent system transition)*** *Let $\phi \longrightarrow \psi$ be an interaction rule, $(\beta, \delta, \iota, \kappa)$ a BDI agent configuration and θ a ground variable substitution w.r.t. ϕ and ψ. Then*

$$\frac{(\beta, \delta, \iota, \kappa) \models \phi\theta, (\beta, \delta, \iota, \kappa) \oplus \psi\theta = (\beta', \delta', \iota', \kappa')}{(\beta, \delta, \iota, \kappa) \longrightarrow (\beta', \delta', \iota', \kappa')}$$

We say that the rule $\phi \longrightarrow \psi$ induces this agent system transition.

Although the semantics of interaction rules is defined for ground query and update formulas, a practical programming system has to handle also non-ground interaction rules. We therefore consider the concept of *grounding on-the-fly*. Non-ground rules are grounded at the time of their execution. For this, a mechanism of variable instantiation similar to the one provided by Prolog can be used. For other languages like C, C++, or Java, mechanism of returning a value to a variable passed as an argument by reference can be exploited.

As the execution of a BDI agent starts with some initial configuration, all other possible transitions are determined by the first one. Thus a BDI agent is formally defined as follows.

Definition 8. ***(BDI agent)*** *A BDI agent is a tuple $(\beta_0, \delta_0, \iota_0, \kappa_0, \mathcal{IR})$, where $(\beta_0, \delta_0, \iota_0, \kappa_0)$ is the initial configuration of an agent and $\mathcal{IR}$ is a set of interaction rules.*

Given an initial configuration of an agent system, we can see its evolution as a path within the transition system. We call such a path *computation run*.

[1]Note, that we consider here an application of a C++ *interpreter*. Therefore a given expression can be executed at any time.

Definition 9. ***(computation run)*** *A computation run $Comp(s_0)$ for a BDI agent $(\beta_0, \delta_0, \iota_0, \kappa_0, \mathcal{IR})$, where $s_0 = (\beta_0, \delta_0, \iota_0, \kappa_0)$, is a finite or infinite sequence $s_0, \ldots, s_n, \ldots$ of BDI agent configurations and $\forall i \geq 0 : s_i \longrightarrow s_{i+1}$ is an agent system transition.*

The semantics of a BDI agent system is the set of all possible computation runs. This corresponds to all possible evolutions of an agent system within its transition system.

Because of non-determinism there are more than one possible evolutions of an agent system. Basically, there are two sources of non-determinism in the BDI agent system:

internal: caused by multiple applicable interaction rules w.r.t. a single configuration, and

external: coming from non-determinism of agent's environment and uncertainty of results of agent's actions in it.

While external non-determinism of an agent system cannot be influenced by an agent itself, the degree of internal is handled by agent system's interpreter.

2.3 Interpreter

What exactly is the relation between the interpreter of an agent system and the set of interaction rules? While interaction rules secure the encoding of the dynamics of the system, the interpreter is responsible for their *selection* and *execution*. In order to achieve the highest possible flexibility, we have to provide a mechanism for the fine-tuning of the interpreter.

Tuning the mechanism of rule selection can be done by means of partial ordering over interaction rules. This gives a very high degree of flexibility and provides enough space for further extensions. It is even possible to remove internal non-determinism of an agent's transition system by applying a total ordering on interaction rules. Such a system would then resemble a standard Prolog interpreter which applies rules simply in order in which they were encoded in the source program. On the other extreme side, when no ordering on interaction rules is applied, the only mechanism governing the evolution of an agent system is the content of query formulas in query parts of interaction rules. Selection of a rule to execute is then arbitrary.

Definition 10. ***(interpreted run)*** *Let $(\beta_0, \delta_0, \iota_0, \kappa_0, \mathcal{IR})$ be a BDI agent and $\prec$ be a partial ordering on $\mathcal{IR}$. We say that the computation run $Comp((\beta_0, \delta_0, \iota_0, \kappa_0))$ is an interpreted run iff for each agent system transition $(\beta_i, \delta_i, \iota_i, \kappa_i) \longrightarrow (\beta_{i+1}, \delta_{i+1}, \iota_{i+1}, \kappa_{i+1})$ induced by a rule $r = \phi \longrightarrow \psi$, r is some minimal rule applicable to configuration $(\beta_i, \delta_i, \iota_i, \kappa_i)$ w.r.t. partial ordering $\prec$.*

Considering only interpreted runs of BDI agent system, BDI agent's definition could be extended by including the ordering $\prec$ to it.

Example 5. ***(running example cont.)*** *Consider an extension of the agent introduced in Examples 1, 2 and 3.*

Let the base of desires of this agent be implemented as a simple set of string identifiers {Load(ContID),Unload(ContID)}, *standing for loading and unloading the cart. The language of desires $\mathcal{L}_D$ is Prolog. Query and update operators correspond to Prolog query execution of predicates* isGoal/1, addGoal/1 *and* removeGoal/1 *with the obvious semantics.*

The base of intentions is implemented as a queue of string identifiers {PerformTake(ContID),PerformPut(ContID)} *standing for raising and putting down the container. Language of intentions $\mathcal{L}_I$ is also Prolog and query/update operations correspond to execution of predicates* currentInt/1, queueInt/1 *and* dequeueInt*/1.*

Finally the language of capabilities $\mathcal{L}_C$ is C and query/update are implemented as for beliefs by invoking the following routines:

```
int isContainerAtLoc(int nContID);
int isCartAtLoc();
void take(int nContID);
void put(int nContID);
```

Sample interaction rules for loading a container are

Q_D(isGoal(Load(ContID)))
  $\longrightarrow$ U_I(queueInt(PerformPut(ContID)))

Q_I(currentInt(PerformPut(ContID))) $\wedge$
Q_B(crane.isHolding(ContID) &&
    crane.isCartPrepared())
  $\longrightarrow$ U_C(put(ContID)) $\circ$
      U_I(dequeueInt(PerformPut(ContID)))

Considering a precedence ordering on the rules above, in one step of the interpreter's cycle, the upper rule will be evaluated as the first. If an agent has the goal to load a container (e.g. Load(2)$\in$*belief base), then this rule will be also executed, which will cause the term* PerformPut(2) *to be appended to the end of the queue of intentions. Then the interpreter will start the cycle all over again. If the query part of the first rule is false, the interpreter will proceed to considering the second interaction rule.*

Note, that in the second rule, we exploited the internal ability of C++ implementation of belief base component to evaluate conjunctions of expressions, instead of using query of the form $Q_B \wedge Q_B$.

By further extending this basic interpreter, by e.g. allowing changes of the ordering of interaction rules on-the-fly, we can implement a very dynamic behaviour of an agent system. However, this is not in the scope of this paper and we will investigate properties of such extensions in future work.

3. AGENT REASONING MODELS

The definition of a BDI agent system as given in the last section is quite abstract: it does not make any differences between individual BDI components. This uniformity is a direct implication of the requirement of independence of BDI agent's components w.r.t. applied knowledge representation technique.

However, we believe that crucial differences of BDI system components can be reflected in *constraints* on their interactions. By allowing, or enforcing certain interactions between components of a BDI agent, we can achieve various properties of the whole system. We will show that a wide range of agent reasoning models can be implemented in our framework by regulating types of interaction rules.

3.1 I-System

Implementations of rational agent reasoning model usually descend from the original I-system [14], which axiomatizes interactions within rational agents. Informally, an agent should adopt only goals it believes to be an option w.r.t. to its beliefs (AI1). An agent should adopt intentions only in order to achieve its goals (AI2). If an agent has an intention to perform a certain action, it will eventually also perform it (AI3). It should be aware of the fact that it committed itself to certain goals and intentions (AI4, AI5). If an agent intends to achieve something, it also has to have a goal to intend it (AI6). It should be aware of its actions, i.e. if it performs certain action, it should also believe that it already performed it (AI7). And finally an agent should never hold its intentions forever, i.e. each intention must be eventually dropped (AI8).

Out of these eight basic axioms, four (AI1-AI3 and AI7) enforce certain interactions between components. These correspond to interaction rules which, following the notation introduced in Example 1, look like $Q_B \longrightarrow U_D$ (AI1), $Q_D \longrightarrow U_I$ (AI2), $Q_I \longrightarrow U_C$ (AI3) and $Q_C \longrightarrow U_B$ (AI7). Axiom AI7 is implemented here indirectly, since agents can learn about their successful actions by perceiving changes caused by them in their environments. An alternative way to implement this axiom is to keep history of actions agent tried to perform in its belief base. We can achieve this by interaction rules of type $Q_I \longrightarrow U_C \circ U_B$ (AI3).

Axioms AI4 and AI5 are secured via accessibility of all BDI components for the agent reasoning mechanism. Interaction rules can examine desire and intention bases by means of queries, so agent reasoning mechanism is "aware" of the content of its components. Axiom AI6 is satisfied by simple execution of an interaction rule of the type $Q_D \longrightarrow U_I$. Finally axiom AI8 is very vaguely specified, since "eventually" can be arbitrarily long. Realisation of this axiom is more a matter of a particular implementation of agent's base of intentions, than a feature of software agent architecture. Finally, rational agents interact with their environment by rules of the form $Q_I \longrightarrow U_C$ for performing actions and $Q_C \longrightarrow U_B$ for perceiving events.

3.2 Extensions of I-System

A closer look at the flow of information in the rational reasoning model, described in the previous subsection, reveals that on the ground of events occurring in the environment, an agent adopts beliefs, which are again the basis for adopting desires. On the ground of desires, intentions are adopted which are finally the base for agent's actions in the environment. This information flow can be informally described as a cycle of BDI components through which the information flows $C \rightarrow B \rightarrow D \rightarrow I \rightarrow C$. By inserting additional links and shortcuts, we can achieve various different agent reasoning models.

In their original paper [14], authors provide examples of *blind*, *single minded*, and *open minded* agent. Informally, a blindly committed agent maintains its intentions until it actually believes that it had achieved them. Single minded agent maintains them as long as it believes they are still possible to achieve and finally open minded agent maintains its intentions only as long as these intentions are still its goals.

In our modular BDI architecture with appropriate implementation of BDI components, a blind agent would need interaction rules for removing intentions from the intention base once it recognizes that actions based on these intentions were successful in the environment. A single minded agent cancels certain intentions with preconditions φ exactly when φ does not hold anymore. Informally, for every rule $Q_B(\varphi) \wedge \ldots \longrightarrow U_I(+\varphi')$, which introduces certain intention into the base of intentions, also the interaction rule $Q_B(\neg\varphi) \longrightarrow U_I(-\varphi')$ must occur in agent's rule base. Similarly for an open minded agent, but instead of Q_B, it would query its base of desires Q_D.

Apart from rational agents, we can also model irrational[2] agent reasoning in our framework. Consider for example a *servant* agent, to which goals can be "implanted" from outside as commands which it should follow, although it doesn't believe they are an option w.r.t. its beliefs. This can be implemented by allowing rules of the type $Q_C \longrightarrow U_D$. Informally, such a rule can be interpreted as *"upon perceiving a command C, agent should carry out C"*. A slightly stricter version of servant agent, is a *slave* agent, to which intentions could be changed and added in a similar way by interaction rule of the type $Q_C \longrightarrow U_I$.

Slave agents resemble more software objects in their traditional meaning in object oriented programming. They simply execute a requested procedure without having a "free will" not to do so. However they are still able to freely decide on later canceling this intention. As an extreme case of irrational reasoning a *religiously fanatic* agent can be considered. Such an agent doesn't want to believe facts which are conflicting its goals, or intentions. According to its current intentions and goals, it is able to change its beliefs by executing rules of the type $Q_D/Q_I \longrightarrow U_B$.

Although purely irrational agents like slave agent lack autonomy and thus do not comply with classic definitions of an agent, mixing rationality with certain degree of non-rational behaviour (w.r.t. certain aspects of agent's functionality) can lead to more efficient implementations of software agent systems. However, this claim has to be yet experimentally demonstrated.

3.3 Modular BDI Programming Platform

As we already mentioned, the main disadvantage of our modular BDI architecture is its high degree of abstraction. On the other hand, we showed that it provides also a high degree of flexibility and a lot of space for customization to particular needs.

Implementation of a software development platform based on our proposal should be based on a plug-in architecture supporting development of BDI components in various programming languages. The interpreter of interaction rules serves only as a glue between them. Each plug-in should be an interpreter for a specific programming language able to evaluate queries and perform updates on the current operational state of the component. Since nowadays many interpreters for various programming languages ranging from C++ to Prolog, LISP or Java are available, such implementation is feasible.

Because of its high flexibility, a programming platform based on our modular BDI architecture could serve as a testbed for investigating applications based on combinations of various knowledge representation techniques. We will pay special attention to applications of logic programming

[2]In the sense of *not rational w.r.t. I-System.*

techniques and especially declarative non-monotonic knowledge representation approaches like Answer Set Programming (ASP) [7] (see also [6] for the relevance of ASP in agent programming). For ASP, there's a vast amount of research on its practical applications (see e.g. [1]) as well as knowledge base updates (e.g. [10]) a lot of which can be used and practically evaluated using our system.

4. RELATED WORK

Because of its legacy, 3APL agent programming language [8] is the closest relative to our modular BDI architecture. Modular BDI schema is rather a generalization of 3APL architecture than a parallel approach. 3APL uses fixed logic based languages for implementation of BDI components. In the 3APL platform [5] Prolog is used to implement the belief base. Goal base is realized as a set of Prolog terms, and plans in the plan base are stack resembling structures of Prolog terms. In the language of our framework, 3APL allows allows only certain types of interaction rules, namely goal rules of the form $Q_D \wedge Q_B \longrightarrow U_D$, plan rules $Q_I \wedge Q_B \longrightarrow U_I$ and interaction rules $Q_D \wedge Q_B \longrightarrow U_D$ [4]. Since belief base of a 3APL agent is implemented in Prolog, all other BDI components are therefore also limited to use structures of first order logic terms.

Another important representative of BDI agent programming systems is AgentSpeak(L) [13] and its incarnation in the agent development platform Jason [3]. AgentSpeak(L) is the original implementation of already discussed I-system in a logic based language. Similarly to 3APL, it uses language of first order logic terms for belief base and stack structures for implementation of intentions. The goal base is realized via direct event processing. Contrary to 3APL, AgentSpeak(L) is even more specific in terms of queries and updates of its components. Interaction rules, or plans, are of the type $Q_D \wedge Q_B \longrightarrow U_I$, or $Q_D \wedge Q_B \longrightarrow U_D$, where updating the base of desires means firing an event. Querying it is similar to processing events from event pool which is updated according to agent's perceptions.

We see the independence of implementation of an individual BDI component of knowledge representation technique as the main advantage of our modular architecture over 3APL and AgentSpeak(L). Because of their focus on rational agent reasoning model, both 3APL and AgentSpeak(L) limit design freedom of a system programmer and enforce particular software development methodology together with a specific knowledge representation technique. Use of proprietary programming language limits 3APL and AgentSpeak(L) w.r.t. exploitation of a wide range of standard libraries for software development and external code packages. This has a strong impact on possibilities of their integration with legacy systems. On the other hand, because of the clear specification of the architecture details, like implementation languages of BDI components, systems built using 3APL or AgentSpeak(L) have probably much clearer theoretical properties w.r.t. verification of an agent system.

The other extreme of the spectrum is represented by the second family of BDI inspired programming frameworks which includes systems like JACK [17] and Jadex [12]. These are focused on integration with a mainstream programming languages (Java in both cases) and provide a high degree of freedom w.r.t. software development techniques used. However, since they both provide a particular methodology for development of internal structure of components of BDI system, integration with other knowledge representation approaches is not straightforward. Finally, they do not provide a clear semantics of systems developed with them, therefore study of theoretical properties of their applications is even more difficult than in the case of our BDI architecture.

Our modular BDI architecture proposal is positioned somewhere between the two approaches to BDI agent programming. On one hand it provides a clear operational semantics and a simple technique for implementation of agent reasoning and on the other it allows high degree of integration with existing programming languages, standard libraries and external legacy code.

5. CONCLUSION AND FUTURE WORK

In this paper we presented a modular BDI architecture which is independent of the underlying knowledge representation approach used and provides a high degree of flexibility for implementation of rational and non-rational agent reasoning models. Its core, interaction rules, are specified as the connecting element between heterogeneous components of BDI agent system. We are convinced that by using such a flexible approach, it will be easier to step from agent system analysis to its implementation and exploit strengths of mainstream programming languages and wide range of libraries provided for them.

Since the proposed modular BDI architecture was conceived to provide support for various, particularly logic based, knowledge representation techniques, our future work will be focused in this direction. We are planning to implement the interpreter for the proposed modular BDI system and a set of generic plug-ins for various knowledge representation approaches. We are interested especially in exploitation of non-monotonic reasoning and incomplete information handling techniques like Answer Set Programming [6, 7]. We will develop a set of plug-ins for programming languages like SMODELS [16] and C++ in order to study properties of agent systems built with them.

The degree of abstraction, our architecture provides, allows embedding various agent reasoning models into it (including those of 3APL or AgentSpeak(L)) and thus provides a common base for their comparison and possibly also classification. Therefore we will also study properties of various agent reasoning models in our BDI architecture and extensions of it with concepts like role of an agent and agency.

6. REFERENCES

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[2] R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations.* Kluwer Academic Publishers, 2005.

[3] R. H. Bordini, J. F. Hübner, and R. Vieira. *Jason and the Golden Fleece of Agent-Oriented Programming*, chapter 1, pages 3–37. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [2], 2005.

[4] M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. C. Meyer. A Programming Language for Cognitive Agents Goal Directed 3APL. In M. Dastani,

J. Dix, and A. E. Fallah-Seghrouchni, editors, *PROMAS*, volume 3067 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2003.

[5] M. Dastani, M. B. van Riemsdijk, and J.-J. Meyer. *Programming Multi-Agent Systems in 3APL*, chapter 2, pages 39–68. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [2], 2005.

[6] J. Dix and T. Eiter. Answer Set Programming and Agents. *AgentLink News*, Vol. 19, 2005.

[7] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[8] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[9] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Summary of an Agent Infrastructure. In T. Wagner and O. Rana, editors, *Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[10] J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers of Artificial Intelligence and Applications.* IOS Press, 2003.

[11] M. Luck, P. McBurney, O. Shehory, and S. Wilmott, editors. *Agent Technology Roadmap: A Roadmap for Agent Based Computing.* University of Southampton on behalf of AgentLink III, September 2005.

[12] A. Pokahr, L. Braubach, and W. Lamersdorf. *Jadex: A BDI Reasoning Engine*, chapter 6, pages 149–174. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [2], 2005.

[13] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. V. de Velde and J. W. Perram, editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.

[14] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In *KR*, pages 473–484, 1991.

[15] A. S. Rao and M. P. Georgeff. An Abstract Architecture for Rational Agents. In *KR*, pages 439–449, 1992.

[16] T. Syrjänen and I. Niemelä. The SMODELS System. In T. Eiter, W. Faber, and M. Truszczynski, editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 434–438. Springer, 2001.

[17] M. Winikoff. *JACK(TM) Intelligent Agents: An Industrial Strength Platform*, chapter 7, pages 175–193. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [2], 2005.