Modular BDI Architecture

Peter Novák, Jürgen Dix

Clausthal University of Technology, Germany

May 11th, 2006
AAMAS'06, Hakodate, Japan

Single BDI Agent Programming

Definition

Creating software systems using design architecture inspired by the Beliefs-Desires-Intentions metaphor (cognitive agents?).

BDI agent system (3 layers):

- *knowledge* - attitudes, mental state, state of environment
- *body* - sensors/effectors $\rightsquigarrow$ environment
- *system dynamics* - reasoning and performing actions

Challenges for programming BDI frameworks:

- *theoretical properties* - insight into system properties, essential for system verification
- *practical applicability* - support of traditional SW development techniques, integration with external systems

Single BDI Agent Programming

Definition

Creating software systems using design architecture inspired by the Beliefs-Desires-Intentions metaphor (cognitive agents?).

BDI agent system (3 layers):

- *knowledge* - attitudes, mental state, state of environment
- *body* - sensors/effectors $\rightsquigarrow$ environment
- *system dynamics* - reasoning and performing actions

Challenges for programming BDI frameworks:

- *theoretical properties* - insight into system properties, essential for system verification
- *practical applicability* - support of traditional SW development techniques, integration with external systems

State-of-the-art

Theoretically driven systems

Declarative agent programming languages built from scratch.

- nice theoretical properties, difficult to integrate with 3rd party systems, declarative knowledge representation (e.g. AGENTSPEAK(L)/JASON, 3APL)

Engineering approaches

Layer of specialized programming constructs over a robust industrial programming language (Java).

- easy to integrate, code re-use, semantics of the underlying language, OOP as a knowledge representation language (e.g. JACK, JADEX)

State-of-the-art

Theoretically driven systems

Declarative agent programming languages built from scratch.

- nice theoretical properties, difficult to integrate with 3rd party systems, declarative knowledge representation (e.g. AGENTSPEAK(L)/JASON, 3APL)

Engineering approaches

Layer of specialized programming constructs over a robust industrial programming language (Java).

- easy to integrate, code re-use, semantics of the underlying language, OOP as a knowledge representation language (e.g. JACK, JADEX)

State-of-the-art

Theoretically driven systems

Declarative agent programming languages built from scratch.

- nice theoretical properties, difficult to integrate with 3rd party systems, declarative knowledge representation (e.g. AGENTSPEAK(L)/JASON, 3APL)

Engineering approaches

Layer of specialized programming constructs over a robust industrial programming language (Java).

- easy to integrate, code re-use, semantics of the underlying language, OOP as a knowledge representation language (e.g. JACK, JADEX)

Problem

State-of-the-art BDI agent programming frameworks take care about too many aspects of the designed system.

Besides providing an agent system dynamics layer, they enforce certain knowledge representation technique.

Solution

We propose a programming system with clear separation between *knowledge representation* and *agent system dynamics*.

Different programming languages are suitable for different knowledge representation tasks.

Focus on agent system dynamics.

Desired properties:

- clear *semantics*
- *modularity* - easy code re-use
- easy *integration* with external/legacy systems

Solution

We propose a programming system with clear separation between *knowledge representation* and *agent system dynamics*.

Different programming languages are suitable for different knowledge representation tasks.

Focus on agent system dynamics.

Desired properties:

- clear *semantics*
- *modularity* - easy code re-use
- easy *integration* with external/legacy systems

Our way to go...

Knowledge Representation:

- encapsulate BDI modules allowing only *query/update* interface
- KR techniques and programming languages ⇝ programmer's decision
- treat agent's capabilities as just another BDI component

Agent System Dynamics:

- interaction between BDI modules ⇝ interaction rules
- application of an interaction rule ⇝ atomic system transition

Our way to go...

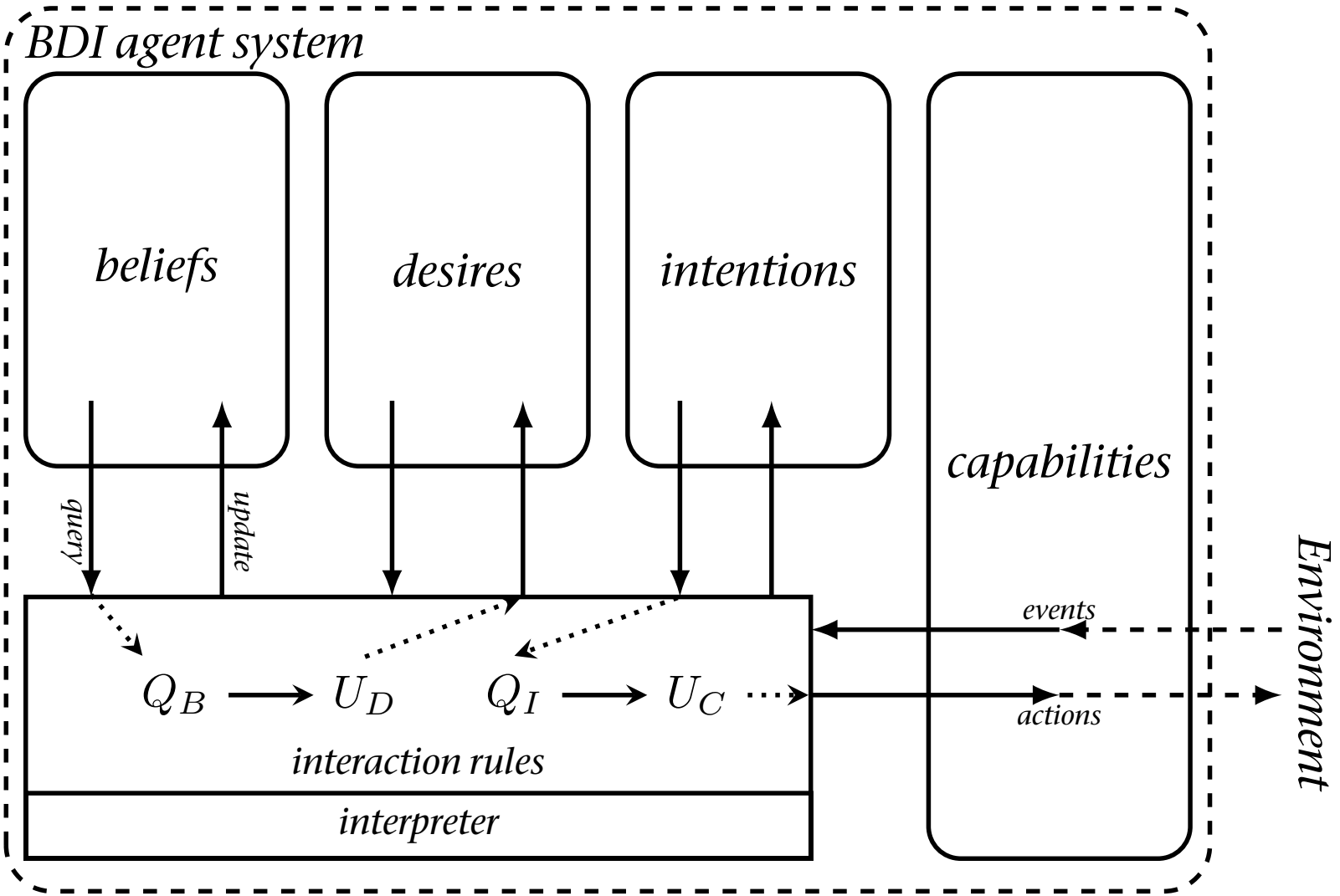
Knowledge Representation:

- encapsulate BDI modules allowing only *query/update* interface
- KR techniques and programming languages $\rightsquigarrow$ programmer's decision
- treat agent's capabilities as just another BDI component

Agent System Dynamics:

- interaction between BDI modules $\rightsquigarrow$ interaction rules
- application of an interaction rule $\rightsquigarrow$ atomic system transition

Architecture



Semantics

Definition

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 and $\mathcal{IR}$ is a set of interaction rules.

- interaction rules have the form $\phi \rightarrow \psi$
- $\mathcal{IR}$ induces a transition system
- interpreter selects and executes interaction rules: evaluate a query and perform a corresponding update
- semantics of a BDI agent is a path within the transition system

Example

Beliefs (Prolog)

```
ready :- cup_present,
         cup_empty,
         not error.
```

Desires (set of Prolog atoms)

```
make_espresso.
```

Intentions (stack - Lisp)

```
(define push ...)
(define pop ...)
(define top? ...)
```

Capabilities (C)

```
void mill_start();
void mill_stop();
int stand_empty();
int cup_empty();
```

$$Q_C(\text{!stand\_empty() \&\& cup\_empty()}) \longrightarrow U_B(\text{assert(cup\_present)})$$

$$Q_B(\text{ready}) \wedge Q_D(\text{make\_espresso}) \longrightarrow U_I(\text{(push (grind boil pour clean))})$$

$$Q_I(\text{(top? grind)}) \longrightarrow U_C(\text{mill\_start()}) \circ U_I(\text{(pop)})$$

Example

Beliefs (Prolog)

```
ready :- cup_present,
         cup_empty,
         not error.
```

Desires (set of Prolog atoms)

```
make_espresso.
```

Intentions (stack - Lisp)

```
(define push ...)
(define pop ...)
(define top? ...)
```

Capabilities (C)

```
void mill_start();
void mill_stop();
int stand_empty();
int cup_empty();
```

$$Q_C(\text{!stand\_empty() \&\& cup\_empty()}) \longrightarrow U_B(\text{assert(cup\_present)})$$

$$Q_B(\text{ready}) \wedge Q_D(\text{make\_espresso}) \longrightarrow U_I(\text{(push (grind boil pour clean))})$$

$$Q_I(\text{(top? grind)}) \longrightarrow U_C(\text{mill\_start()}) \circ U_I(\text{(pop)})$$

Example

Beliefs (Prolog)

```
ready :- cup_present,
         cup_empty,
         not error.
```

Desires (set of Prolog atoms)

```
make_espresso.
```

Intentions (stack - Lisp)

```
(define push ...)
(define pop ...)
(define top? ...)
```

Capabilities (C)

```
void mill_start();
void mill_stop();
int stand_empty();
int cup_empty();
```

$$Q_C(\text{!stand\_empty() \&\& cup\_empty()}) \longrightarrow U_B(\text{assert(cup\_present)})$$

$$Q_B(\text{ready}) \wedge Q_D(\text{make\_espresso}) \longrightarrow U_I(\text{(push (grind boil pour clean))})$$

$$Q_I(\text{(top? grind)}) \longrightarrow U_C(\text{mill\_start()}) \circ U_I(\text{(pop)})$$

Example

Beliefs (Prolog)

```
ready :- cup_present,
         cup_empty,
         not error.
```

Desires (set of Prolog atoms)

```
make_espresso.
```

Intentions (stack - Lisp)

```
(define push ...)
(define pop ...)
(define top? ...)
```

Capabilities (C)

```
void mill_start();
void mill_stop();
int stand_empty();
int cup_empty();
```

$$Q_C(\text{!stand\_empty() \&\& cup\_empty()}) \longrightarrow U_B(\text{assert(cup\_present)})$$

$$Q_B(\text{ready}) \wedge Q_D(\text{make\_espresso}) \longrightarrow U_I(\text{(push (grind boil pour clean))})$$

$$Q_I(\text{(top? grind)}) \longrightarrow U_C(\text{mill\_start()}) \circ U_I(\text{(pop)})$$

Questions?

Thank you for your attention.

Come and visit our poster for more details.