Modular agent programming language

(preliminary report)

Peter Novák, Jürgen Dix

Computational Intelligence Group
Clausthal University of Technology

June 27th, 2006, Dagstuhl

1 Ultimate vision & state-of-the-art

2 Modular BDI Architecture

3 Jazyk - The Language

4 Conclusion

Quest for a practical agent programming system

- *clear semantics* (insight into theoretical properties ⇝ verification?)
- standard software engineering support
  - *modularity* (code re-use, structural decomposition)
  - *expressive syntax* (should be as simple as possible!)
  - *easy integration* with external systems (environment, legacy subsystems, middleware, sensors/actuators)
- design freedom:
  - *choice of KR techniques* (*"different programming languages are good for different KR tasks"*)
  - *deliberation cycle control* - integrated and powerful
  - *bad design* - freedom to implement software in a "wrong" way

What makes a BDI agent program today?

Class of systems with a clear semantics (3APL, AgentSpeak(L), ...):

- 3 modules (knowledge bases) ← enforce fixed KR technique!
- "reasoning" rules (goals-2-actions decomposition) ← constrain system interactions!
- agent's actions specification ← foreign programming language!
- deliberation cycle customization ← associated language, not an integral part of the agent program!

Do we know how to use our agent programming languages?

Can we do better? (our attempt)

abstract agent architecture ➔ programming language.

Abstract architecture - generalize, generalize, generalize!:

- separate KR issues and system dynamics
- component based design (basic set of BDI-inspired components)

Programming language - one agent = one program:

- structural decomposition support
- simple, yet powerful deliberation cycle control $\leadsto$ integral part of the agent program
- IDE, middleware, interaction, ... ← not a primary concern of a programming language!

Modular BDI Architecture

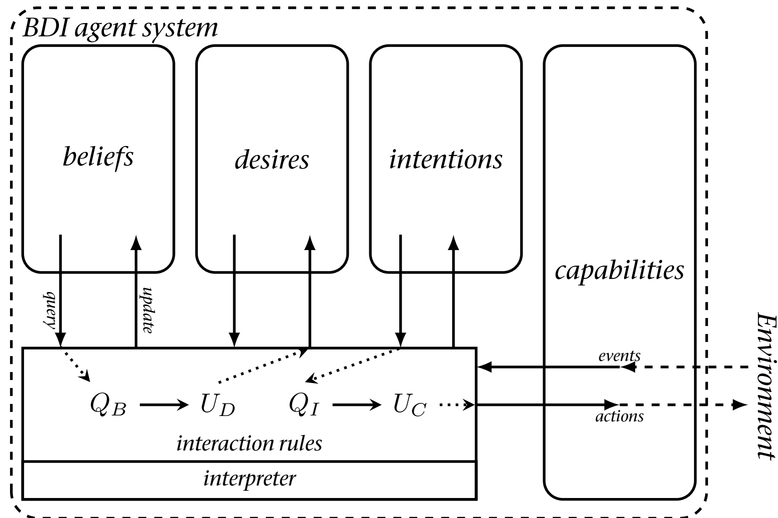
Knowledge Representation:

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

Agent System Dynamics:

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

Architecture



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();
```

$$Q_C(\text{!stand_empty() \&\& cup_empty()}) \longrightarrow U_B(\text{assert(cup_present)})$$

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

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

So what? What is it good for?

Advantages:

- abstract meta-framework for building agent programming languages
  - allows to implement various models of rationality
  - AgentSpeak, 3/2APL & Co. can be seen as instances of this framework
- the least common ground for APLs? (Koen's talk)

Shortcomings:

- extremely abstract way of thinking about agent program
  - if we forget about the purpose of the particular module, the whole thing falls apart ← programer's concern!
  - the same for constraints on rule types allowed
- too poor common ground? (Koen's talk)

Jazyk -The Language

Basic statement:

when query <module> [{. . . }] then update <module> [{. . . }];

```
when
    query desires [{ make_espresso }] and query beliefs [{ ready }]
then
   update intentions [{ (push (grind boil pour clean) }];
```

Modules declaration:

```
declare module beliefs [{
    . . .
    include(myfile.cpp);
    . . .
}];
```

Jazyk - Adding variables

Adding variables:

```
when
    query desires(Type) [{ make(Type) }] and
    query intentions [{grind}] and
    query beliefs(Type,Amount) [{ receipt(Type,Amount) }]
then
   update capabilities(Amount) [{ grind(Amount) }];
```

Semantics similar to Prolog-style free variable binding - evaluated from left to right!

Jazyk - Adding structure decomposition

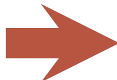
Nested rules:

```
when
    query beliefs [{ needsCleaning }]
then {
    when query beliefs [{ not standEmpty }]
    then update capabilities [{ displayMessage('remove the cup!') }];

    when query beliefs [{ not error }] {
        update capabilities [{ rinse }];
        update beliefs [{ assert(rinsing) }];
    }
}
```

Translation to a basic statement:

```
when <Query1> then {
    when <Query2>
    then <Update2>;
}
```

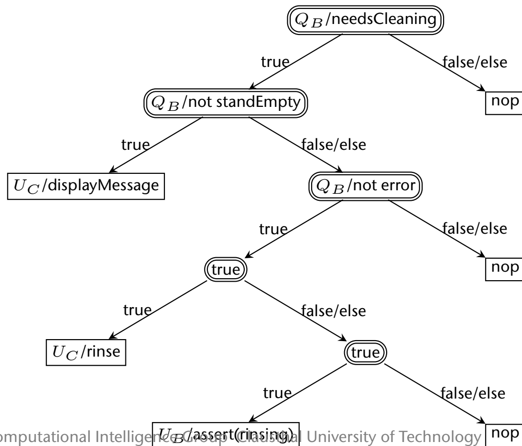


```
when <Query1> and <Query2>
then <Update2>;
```

Deliberation cycle

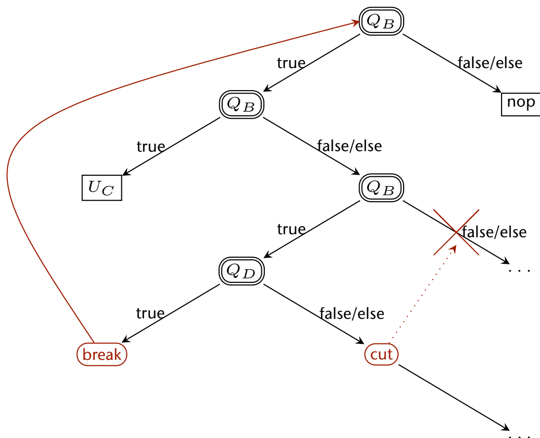
Nested rules induce a tree structure:

depth-first backtrack interpretation!



Interpreter cycle control

Prolog-style deliberation cycle control constructs: cut, break, (try-catch?)



Jazyk - Enhancing modularity

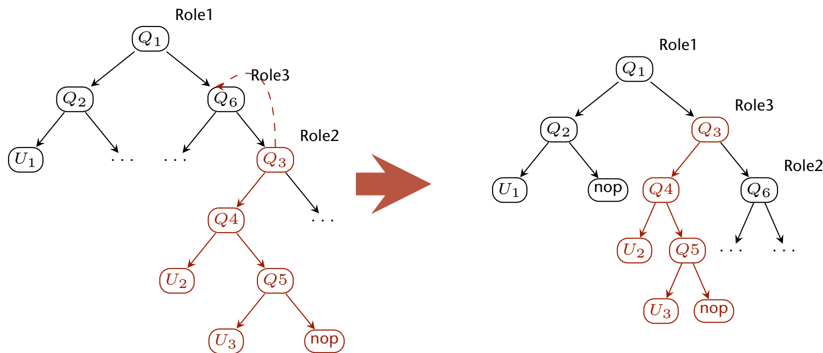
Source code level modularity support: define, apply

```
define CleanMachine : when
      query beliefs [{ needsCleaning }]
  then {
      when query beliefs [{ not standEmpty }]
      then update capabilities [{ displayMessage('remove the cup!') }];

      when query beliefs [{ not error }] {
          update capabilities [{ rinse }];
          update beliefs [{ assert(rinsing) }];
      }
  }
. . .
when query beliefs [{ isIdle }] then apply CleanMachine;
. . .
```

Jazyk - Reflective features

Large subtrees ⇝ roles/behaviors!
Ordering of roles dynamically changes during agent's execution.



On-going and future work

- full fledged interpreter
- modules for Prolog and Lisp
- module for Smodels - Answer Set Programming support integration
- experiments, case study $\rightsquigarrow$ polishing the language

Questions?

Thanks for your attention.