Syntax and Operational Semantics of 2APL

Mehdi Dastani

Utrecht University

September 8, 2014

Abstraction in Multi-Agent Systems

- Individual Agent Level: Autonomy, Situatedness, Proactivity
  - Cognitive concepts: beliefs, goals, plans, actions
  - Deliberation and control: sense/reason/act, reactive/pro-active
- Multi-Agent Level: Social and Organizational Structures
  - Roles: functionalities, activities, and responsibilities
  - Organizational Rules: constraints on roles and their interactions
  - Organizational Structures: topology of interaction patterns and the control of activities
- Environment: Resources and Services that MAS can access and control

2APL: Data Structures and Operations

Data Structures to represent agent mental state

- Beliefs : Information available to agent
- Goals : Objectives that agent want to reach
- Events : Observations of (environmental) changes
- Capabilities : Actions that agent can perform
- Plans : Procedures to achieve objectives
- Reasoning rules : Reason about goals and plans
  - planning rules (goal $\rightarrow$ plan)
  - procedural rules (events $\rightarrow$ plan)
  - plan repair rules (plan $\rightarrow$ plan)

2APL: Data Structures and Operations

Programming Instructions to process mental states

- Generate Plans for Received Events
- Generate Plans for Goals
- Process Exceptions and Handle Failures
- Repair Plans
- Select Plans for Execution
- Execute Plans

Agent Interpreter or Agent Deliberation is a loop consisting of such instructions. The loop determines the behavior of the agent.

Part I

2APL: Syntax

Initializing MAS: Agents and Environments

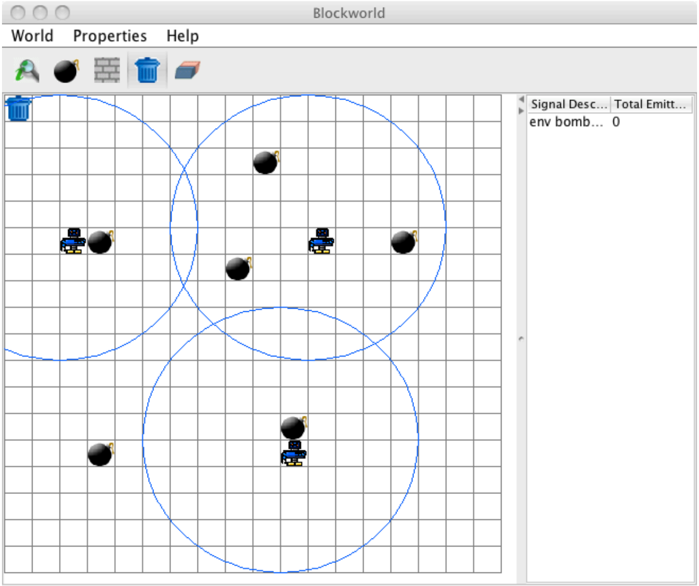
```
<apaplmas>
    <agent name="w1" file="worker.2apl"/>
    <agent name="w2" file="worker.2apl"/>
    <agent name="m" file="manager.2apl">
        <beliefs file="nameWorkers.pl" />
    </agent>

    <environment name="blockworld" file="blockworld.jar">
        <parameter key="gridWidth" value="18"/>
        <parameter key="gridHeight" value="18"/>
    </environment>
</apaplmas>
```

Programming Individual Agents

$$
\begin{array}{lll}
\langle \mathit{Program} \rangle & ::= & \{\text{"Include:"}\ \langle \mathit{ident} \rangle \\
 & \mid & \text{"Beliefupdates:"}\ \langle \mathit{BelUpSpec} \rangle \\
 & \mid & \text{"Beliefs:"}\ \langle \mathit{beliefs} \rangle \\
 & \mid & \text{"Goals:"}\ \langle \mathit{goals} \rangle \\
 & \mid & \text{"Plans:"}\ \langle \mathit{plans} \rangle \\
 & \mid & \text{"PG-rules:"}\ \langle \mathit{pgrules} \rangle \\
 & \mid & \text{"PC-rules:"}\ \langle \mathit{pcrules} \rangle \\
 & \mid & \text{"PR-rules:"}\ \langle \mathit{prrules} \rangle\ \}
\end{array}
$$

An Example



Blockworld
World
Properties
Help
Signal Desc...
Total Emitt...
env bomb...
0

Programming Individual Agents: An Example

Cleaning Environment

```
Beliefs:
  trap(0, 0).
  clean( blockWorld ) :- not bomb(X,Y) , not carry(bomb).

BeliefUpdates:
   {carry(bomb)}     Drop()          {not carry( bomb)}
   {not carry(bomb)} PickUp()        {carry( bomb)}

Plans:
  startup(0, 1, blue);

Goals:
  clean( blockWorld )
```

Beliefs, Goals, Plans and Updates

$$
\begin{array}{lcl}
\langle \mathit{beliefs} \rangle & ::= & \langle \mathit{belief} \rangle + \\
\langle \mathit{belief} \rangle & ::= & \langle \mathit{ground\_atom} \rangle \text{ "."} \\
 & | & \langle \mathit{atom} \rangle \text{ ":}-\text{"} \ \langle \mathit{literals} \rangle \text{"."} \\
 & & \\
\langle \mathit{goals} \rangle & ::= & \langle \mathit{goal} \rangle + \\
\langle \mathit{goal} \rangle & ::= & \langle \mathit{ground\_atom} \rangle \ \{ \text{ "and" } \langle \mathit{ground\_atom} \rangle \ \} \\
 & & \\
\langle \mathit{plans} \rangle & ::= & \langle \mathit{plan} \rangle + \\
 & & \\
\langle \mathit{BelUpSpec} \rangle & ::= & (\text{ "\{"} \langle \mathit{belquery} \rangle \text{ "\}" } \langle \mathit{belUp} \rangle \text{"\{"} \langle \mathit{literals} \rangle \text{"\}"} \ )+
\end{array}
$$

Programming Individual Agents: An Example

Cleaning Environment

```
Beliefs:
  trap(0, 0).
  clean( blockWorld ) :- not bomb(X,Y) , not carry(bomb).

BeliefUpdates:
   {carry(bomb)}     Drop()          {not carry( bomb)}
   {not carry(bomb)} PickUp()        {carry( bomb)}

Plans:
  startup(0, 1, blue);

Goals:
  clean( blockWorld )
```

Plans and Actions

$\langle plan \rangle$	::=	"skip" | $\langle belUp \rangle$ | $\langle dirBelUp \rangle$ | $\langle test \rangle$
	|	$\langle abstractaction \rangle$
	|	$\langle adoptgoal \rangle$ | $\langle dropgoal \rangle$
	|	$\langle externalaction \rangle$ | $\langle sendaction \rangle$
	|	$\langle whileplan \rangle$ | $\langle ifplan \rangle$
	|	$\langle sequenceplan \rangle$ | $\langle atomicplan \rangle$
$\langle dirBelUp \rangle$	::=	("+" | "-") $\langle atom \rangle$
$\langle test \rangle$	::=	"B(" $\langle belquery \rangle$ ")" | "G(" $\langle goalquery \rangle$ ")"
	|	$\langle test \rangle$ & $\langle test \rangle$
$\langle externalaction \rangle$	::=	"@" $\langle ident \rangle$ "(" $\langle atom \rangle$ "," $\langle Var \rangle$ ")"
$\langle sendaction \rangle$	::=	"Send(" $\langle iv \rangle$ "," $\langle iv \rangle$ "," $\langle atom \rangle$ ")"

Composite Plans

$$\langle \textit{whileplan} \rangle ::= \text{"while"} \ \langle \textit{test} \rangle \ \text{"do"} \ \langle \textit{scopeplan} \rangle$$

$$\langle \textit{ifplan} \rangle ::= \text{"if"} \ \langle \textit{test} \rangle \ \text{"then"} \ \langle \textit{scopeplan} \rangle \ [\text{"else"} \ \langle \textit{scopeplan} \rangle]$$

$$\langle \textit{sequenceplan} \rangle ::= \langle \textit{plan} \rangle \ \text{";"} \ \langle \textit{plan} \rangle$$

$$\langle \textit{scopeplan} \rangle ::= \text{"\{"} \ \langle \textit{plan} \rangle \ \text{"\}"}$$

$$\langle \textit{atomicplan} \rangle ::= \text{"["} \ \langle \textit{plan} \rangle \ \text{"]"}$$

Reasoning Rules

$$
\begin{array}{lll}
\langle pgrules \rangle & ::= & \langle pgrule \rangle + \\
\langle pgrule \rangle & ::= & [\langle goalquery \rangle]\ "<-"\ \langle belquery \rangle\ "|"\ \langle plan \rangle \\
\\
\langle pcrules \rangle & ::= & \langle pcrule \rangle + \\
\langle pcrule \rangle & ::= & \langle atom \rangle\ "<-"\ \langle belquery \rangle\ "|"\ \langle plan \rangle \\
\\
\langle prrules \rangle & ::= & \langle prrule \rangle + \\
\langle prrule \rangle & ::= & \langle planvar \rangle\ "<-"\ \langle belquery \rangle\ "|"\ \langle planvar \rangle
\end{array}
$$

Programming Individual Agents: An Example

Cleaning Environment

```
PG-rules:
  clean( blockWorld ) <- bomb( X, Y ) |
  {
    goto( X, Y ); @blockworld( pickup( ), _ ); PickUp();
    -bomb( X, Y ); goto(0, 0); @blockworld( drop( ), _ ); Drop();
  }


PC-rules:
  message( sally, inform, La, On, bombAt( X, Y ) ) <- true |
  {
    if B( not bomb( A, B ) ) { +bomb(X, Y); adoptz( clean( blockWorld ) ); }
    else { +bomb( X, Y ); }
  }


PR-rules:
  @blockworld( pickup(), _ ); REST; <- true |
  {
    @blockworld( sensePosition(), POS ); B(POS = [X,Y]); -bomb(X, Y );
  }
```

2APL Environment: Java-based Environment API

- `Environment` base class
- implementing actions as methods
  - inside action methods external events can be generated to be perceived by agents as percepts

```
package blockworld;

public class Env extends apapl.Environment {


   public void enter(String agent, Term x, Term y, Term c){...}

   public Term sensePosition(String agent){...}

   public Term pickup(String agent){...}

   public void north(String agent){...}

  ...

}
```

Part II

Operational Semantics

Operational Semantics

Meaning of programming language explained by operational semantics:

- defines computation steps a program configuration may make
- $C \rightarrow C'$: means configuration C evolves into configuration C'
- $\frac{P}{C \rightarrow C'}$: if premise P holds transition $C \rightarrow C'$ can be derived

Benefits of operational semantics:

- study programming constructs in a rigorous manner
- facilitates proving general properties about language
- close to the implementation of an interpreter
- facilitates model checking

2APL Semantics: Configuration

- Multi-Agent Configuration: $\langle \{A_1, \ldots, A_n\}, \chi \rangle$
- Individual Agent Configuration: $A_i = \langle i, \sigma_i, \gamma_i, \Pi_i, \theta_i, \xi_i \rangle$
- Transitions are derived by Transition Rules
  - A transition is possible if certain conditions hold

$$\frac{\text{Condition}}{C \rightarrow C'}$$

  - A transition is possible if another transition is possible

$$\frac{C_1 \rightarrow C_1'}{C_2 \rightarrow C_2'}$$

Belief Update Actions

The successful execution of a belief update action α modifies the belief and goal bases.

$$\frac{T(\alpha\theta, \sigma) = \sigma'}{\langle \iota, \sigma, \gamma, \{(\alpha, id)\}, \theta, \xi \rangle \longrightarrow \langle \iota, \sigma', \gamma', \{\}, \theta, \xi \rangle}$$

Where $\gamma' = \gamma - \{\phi \in \gamma \mid \sigma' \models \phi\}$

Adopt Goals

The successful execution of a goal adopt action adopta(g) adds the goal to the beginning of the goal base.

$$\frac{\sigma \not\models_b g\theta}{\langle \iota, \sigma, [\gamma_1, \ldots, \gamma_n], \{(\mathtt{adopta(g)}, id)\}, \theta, \chi, \xi\rangle \longrightarrow \langle \iota, \sigma, [g\theta, \gamma_1, \ldots, \gamma_n], \{\}, \theta, \chi, \xi\rangle}$$

Applying Planning Goal Rules

A PG-rule κ<-β|π can be applied, if κ is entailed by one of the agent's goals, β is entailed by the agent's belief base.

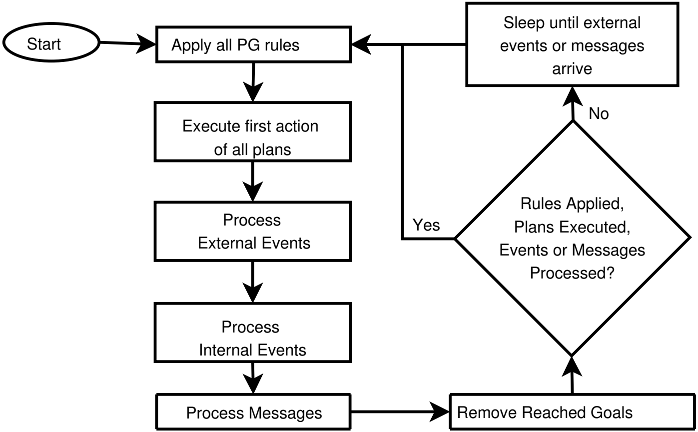
$$\frac{\gamma \models_g \kappa\tau_1 \;\;\&\;\; \sigma \models \beta\tau_1\tau_2}{\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle \longrightarrow \langle \iota, \sigma, \gamma, \Pi \cup \{(\pi\tau_1\tau_2 \;,\; id)\}, \theta, \xi \rangle}$$

Where *id* is a fresh plan identifier.

Part III

Interpreter

Generic BDI Architecture



2APL Interpreter: Deliberation Cycle

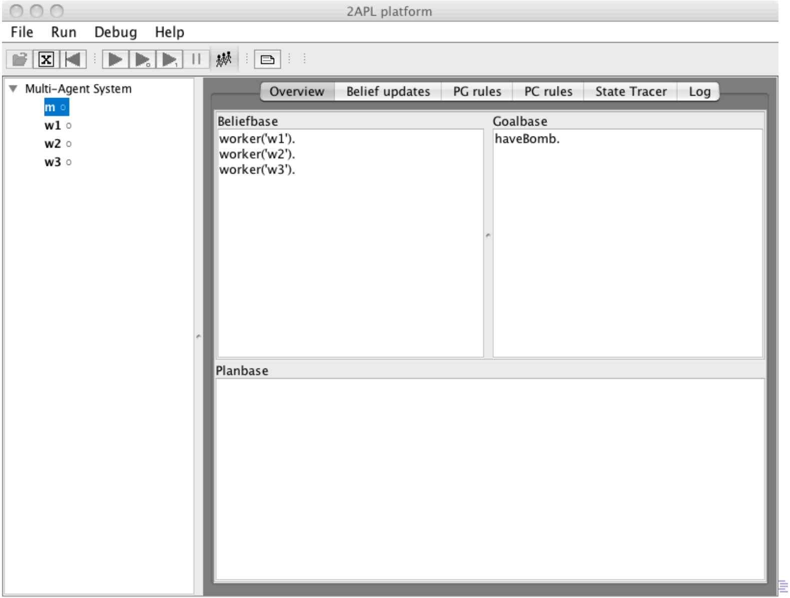
Repeat

- Apply PG-rules
- For each internal event, find and apply a PR-rule
- For each message and external event, find and apply a PC-rule
- Execute one step for each plan

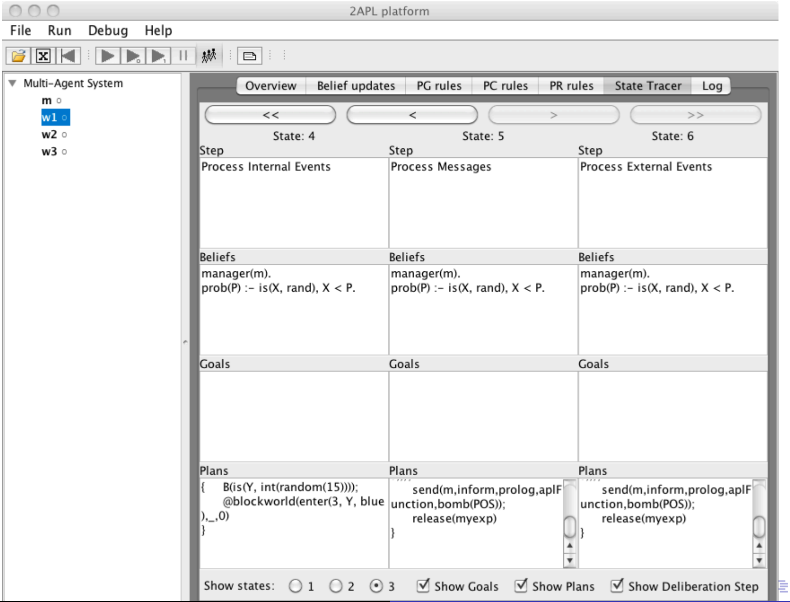
Part IV

Integrated Development Environment

2APL Integrated Development Environment



2APL Integrated Development Environment



Part V

Extension: 2APL Modularity

Modularity in BDI-based Agent Programming

- Modularity is an essential principle in structured programming. It structures a computer program in separate modules.
- Modularization can be used for information hiding and reusability.
- Modularization in existing BDI-based Agent programming languages is to structure an individual agent's program in separate modules, each encapsulating cognitive components.

Modularity: Our Vision

- Roles are functionalities to handle specific situations. They can be specified in terms of BDI concepts.
- An agent profile can be specified in terms of BDI concepts.
- A module represents a BDI state on which it can deliberated. A BDI agent is a deliberation process starting with a BDI state.
- 2APL provides a set of programming constructs to instantiate modules and to change the focus of deliberation at run time.

Modular 2APL: Syntax

$\langle$*2APL_Module*$\rangle$	::=	"Beliefupdates: $\langle$*BelUpSpec*$\rangle$"
	|	"Beliefs:" $\langle$*belief*$\rangle$
	|	"Goals:" $\langle$*goals*$\rangle$
	|	"Plans:" $\langle$*plans*$\rangle$
	|	"PG-rules:" $\langle$*pgrules*$\rangle$
	|	...
...		
$\langle$*plan*$\rangle$	::=	... | $\langle$*createaction*$\rangle$ | $\langle$*releaseaction*$\rangle$ | $\langle$*moduleaction*$\rangle$
...		
$\langle$*createaction*$\rangle$	::=	"create(" $\langle$*ident*$\rangle$ "," $\langle$*ident*$\rangle$ ")"
$\langle$*releaseaction*$\rangle$	::=	"release(" $\langle$*ident*$\rangle$ ")"
$\langle$*moduleaction*$\rangle$	::=	$\langle$*ident*$\rangle$ "." $\langle$*maction*$\rangle$
...		
$\langle$*maction*$\rangle$	::=	"execute(" $\langle$*condition*$\rangle$ ")"
	|	"updateBB(" $\langle$*belief*$\rangle$ ")"
	|	"adopt(" $\langle$*goal*$\rangle$ ")"

Modular 2APL: An Example

```
Beliefs:
  manager(m).

PC-rules:
  message(A, request, play(explorer)) <- manager(A) |
  {
    create(explorer, myexp);
    myexp.execute( B(gold(POS)) );
    send(A, inform, gold(POS));
    release(myexp)
  }

  message(A, request, play(carrier, POS)) <- manager(A) |
  {
    create(carrier, mycar);
    mycar.updateBB( gold(POS) );
    mycar.execute( B(done) );
    send(A, inform, done(POS))
    release(mycar)
  }
```

Features of 2APL: A Summary

- Programming Constructs
  - Multi-Agent System Which and how many agents to create? Which environments? Which agent can access which environment?
  - Individual Agent Beliefs, Goals, Plans, Events, Messages
- Programming Principles and Techniques
  - Abstraction Procedures and Recursion in Plans
  - Error Handling Plan Failure and their revision by Internal Events, Execution of Critical Region of Plans
  - Legacy Systems Environment and External Actions
  - Encapsulation Including 2APL files in other 2APL files
  - Autonomy Adjustable Deliberation Process

Features of 2APL: A Summary

- Integrated Development Environment
  - 2APL platform is Built on JADE and uses related tools
  - Editor with High-Lighting Syntax
  - Monitoring mental attitudes of individual agents, their reasoning and communications
  - Executing in one step or continuous mode
  - Visual Programming of the Deliberation Process

Conclusion and Future works

- 2APL provides a variety of distinguished concepts: Beliefs, Goals, Events, Plans, plan repairs, etc.
- 2APL has an complete operational semantics.
- Logics are developed to verify 2APL programs.
- 2APL comes with an implemented framework that facilitates the execution of multi-agent programs.
- 2APL has an Eclipse Plug-in with colored editors and other IDE facilities such as debugging.
- 2APL supports a strong notion of BDI modularity.
- Integrating different Goal Types in the interpreter.