Agent-based Computing
Jadex: A BDI Reasoning Engine

Maciej Gawinecki

Overview

- Theoretical foundation of BDI
- Introduction to Jadex reasoning engine
- JADE example
- Developing tools in Jadex
- Implementation in Jadex
- Conclusions

Overview

- Theoretical foundation of BDI
- Introduction to Jadex reasoning engine
- JADE example
- Developing tools in Jadex
- Implementation in Jadex
- Conslusions

Theoretical foundation of BDI

- Reasons
  - M. E. Bratman, D. J. Isreal, and M. E. Pollack (1987) *"Plans and resource-bounded practical reasoning."*
  - A. S. Rao, M. P. Georgeff, (1995), *“BDI Agents: From Theory to Practice.”*

BDI abstraction

- Deciding on which goals to achieve and how to achieve them
  - Beliefs: the information an agent has about its surroundings
  - Desires: the things that an agent would like to see achieved
  - Intentions: the desires that an agent is working on; also involves a deeper personal commitment
- Example:
  - Belief: *My students are unhappy...*
  - Desire: *I want to make my students happy.*
  - Intention: *I will buy 22" LCD for each of them!*

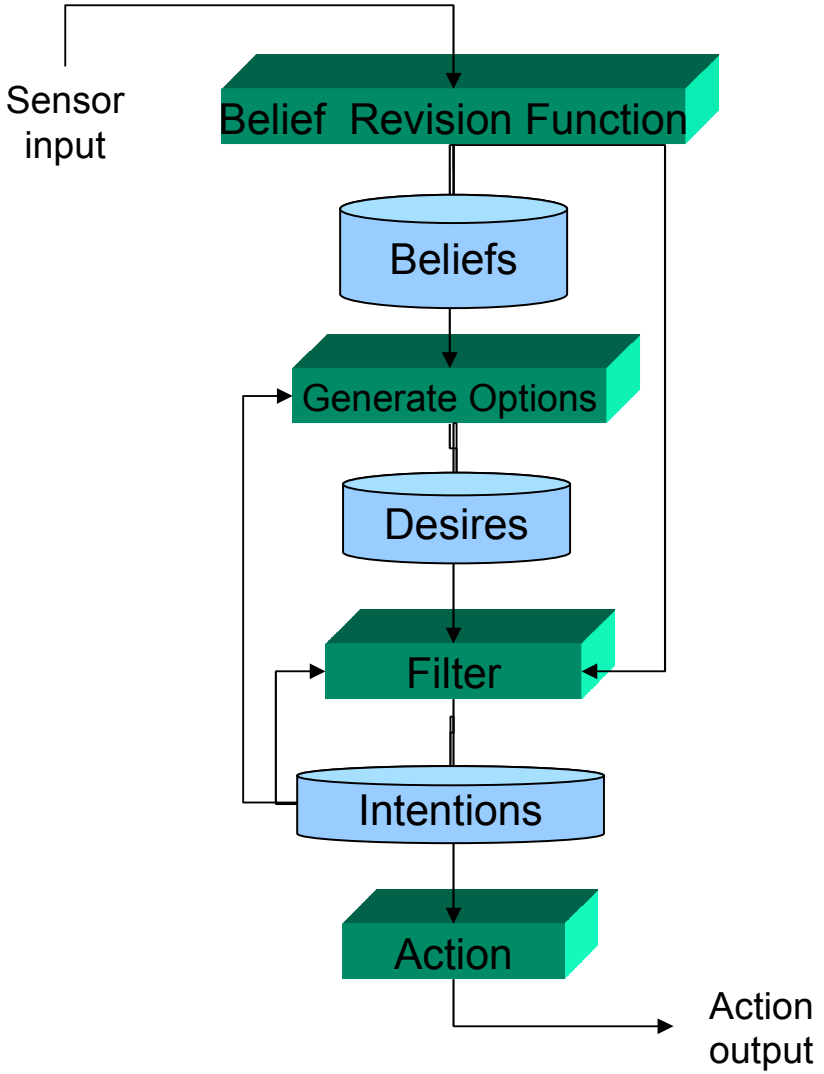
Requirements for BDI Architecture

- A BDI architecture addresses how beliefs, desires and intentions are represented, updated, and processed

- In BDI architecture an agent should (Bratman et al. 1987):
  - monitor its plans when it changes its beliefs,
  - check compatibility with prior plans (*intentions*),
  - propose new plans when enviroments changes.

- These processes should be performed in timely fashion (Bratman et al. 1987).

Generic BDI Architecture



- Generating options and filtering options are together called deliberation

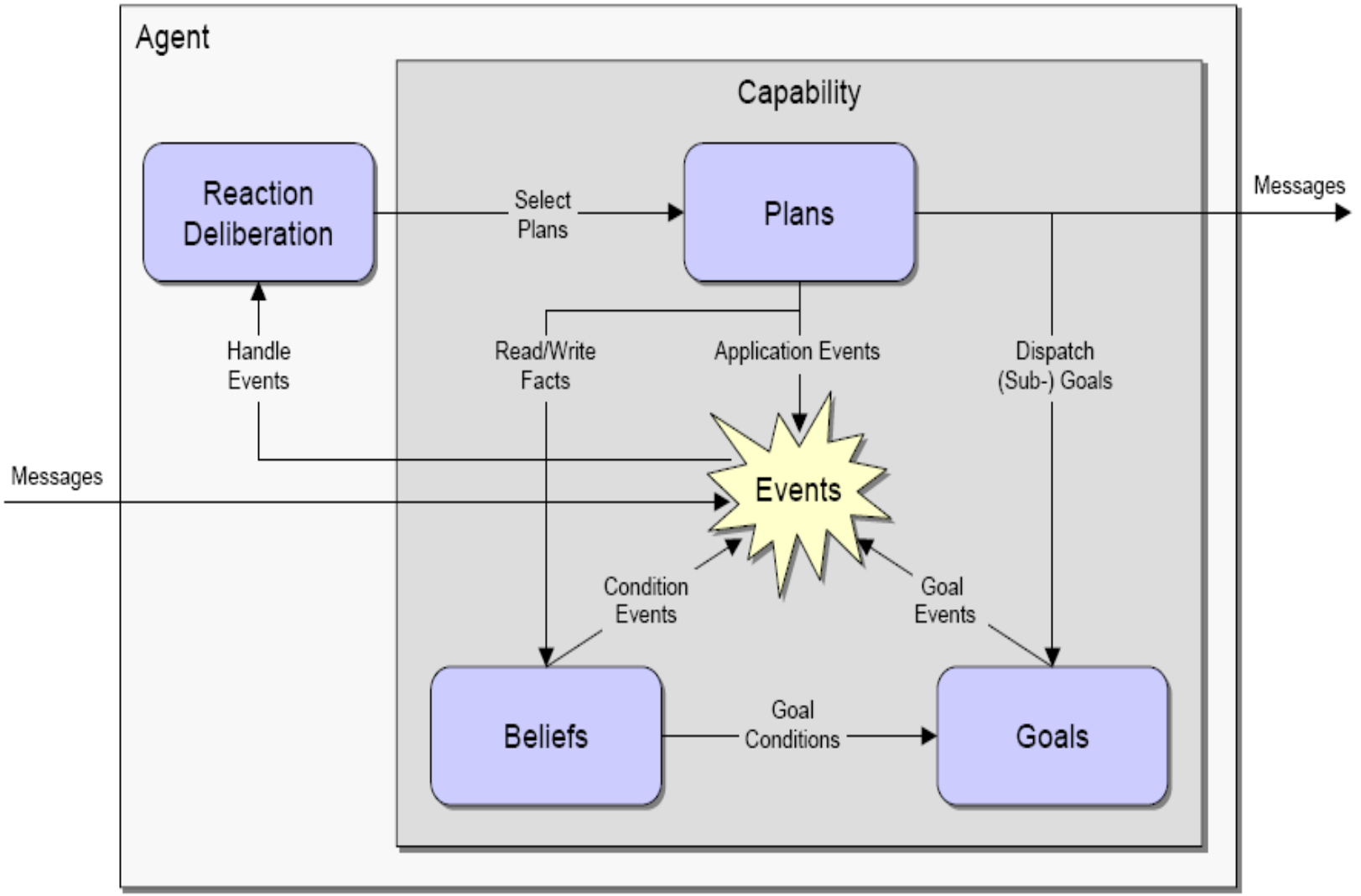
Overview

- Theoretical foundation of BDI
- Introduction to Jadex reasoning engine
- JADE example
- Developing tools in Jadex
- Implementation in Jadex
- Conslusions

Background and Motivation

- Jadex is based on the BDI model
- Integrates agent theories with object-orientation and XML descriptions
- Object-oriented representation of BDI concepts
- Explicit representation of goals allows reasoning about (manipulation of) goals
- Jadex is based on JADE Platform

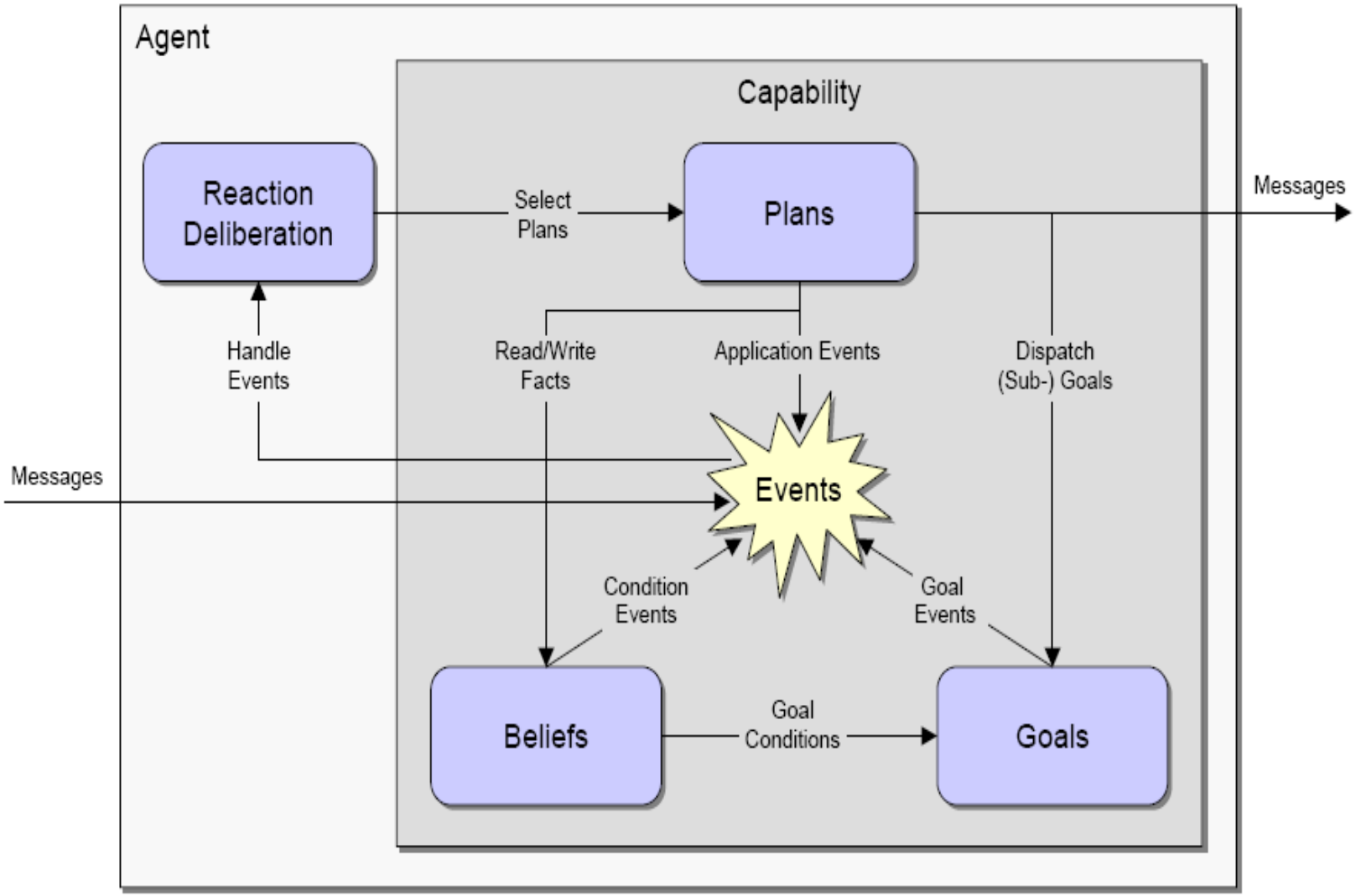
Jadex Abstract Agent Architecture



Beliefs

- Beliefbase contains the knowledge of an agent
  - Beliefs (single facts stored as Java objects)
  - Beliefsets (sets of facts as Java objects)
  - object-oriented representation
- No support for logical reasoning
- Advantages of storing information as facts
  - Central place for knowledge (accessible to all plans)
  - Allows queries over agent‘s beliefs
  - Allows monitoring of beliefs and conditions (e.g. to trigger events / goals)

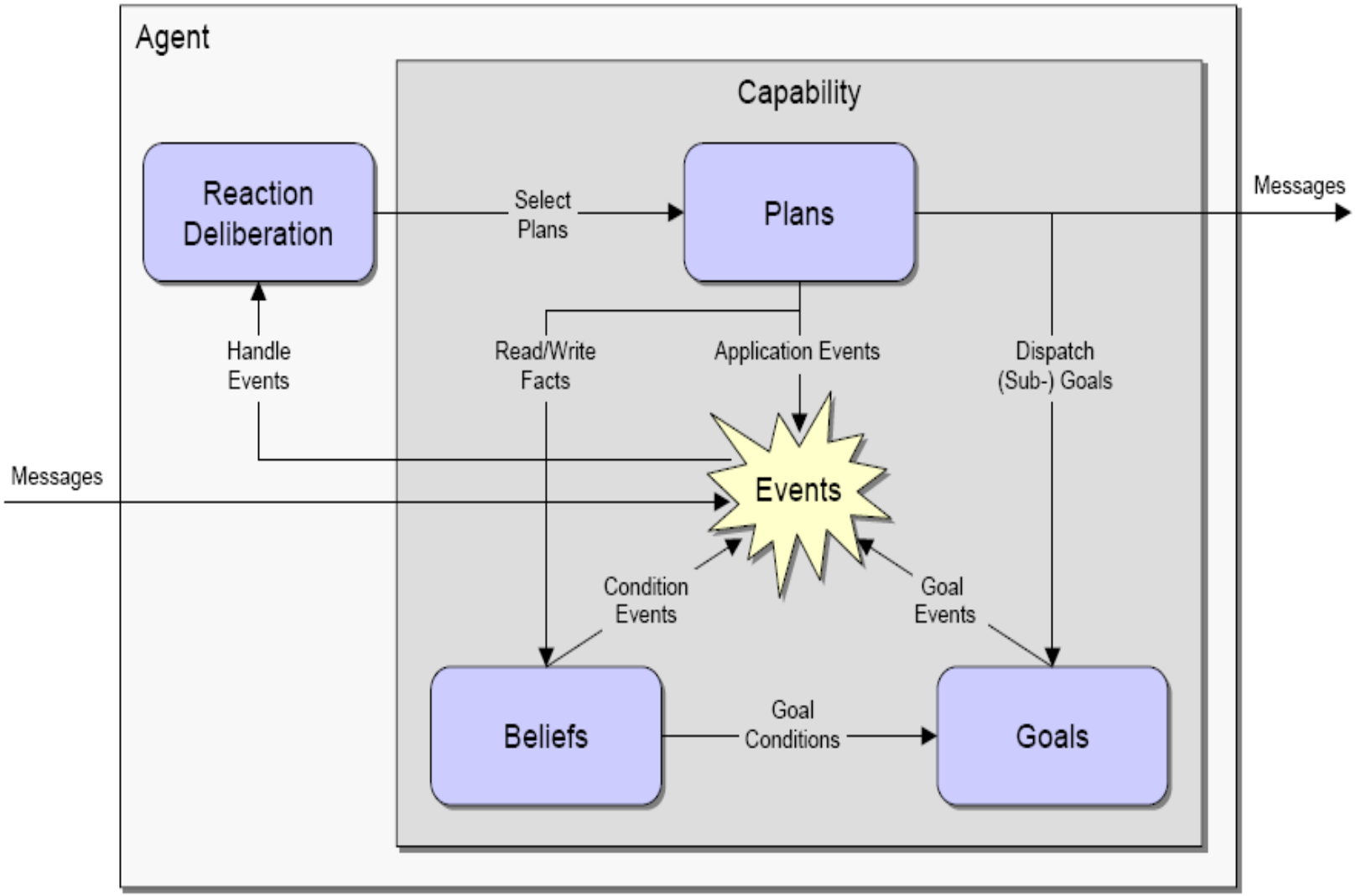
Jadex Abstract Agent Architecture



Goals *(desires)*

- Generic goal types
  - perform (some action)
  - achieve (a specified world state)
  - query (some information)
  - maintain (reestablish a specified world state whenever violated)
- Are strongly typed with
  - name, type, parameters
  - BDI-flags enable non-default goal-processing
- Goal creation/deletion possibilities
  - initial goals for agents
  - goal creation/drop conditions for all goal kinds
  - top-level / subgoals from within plans

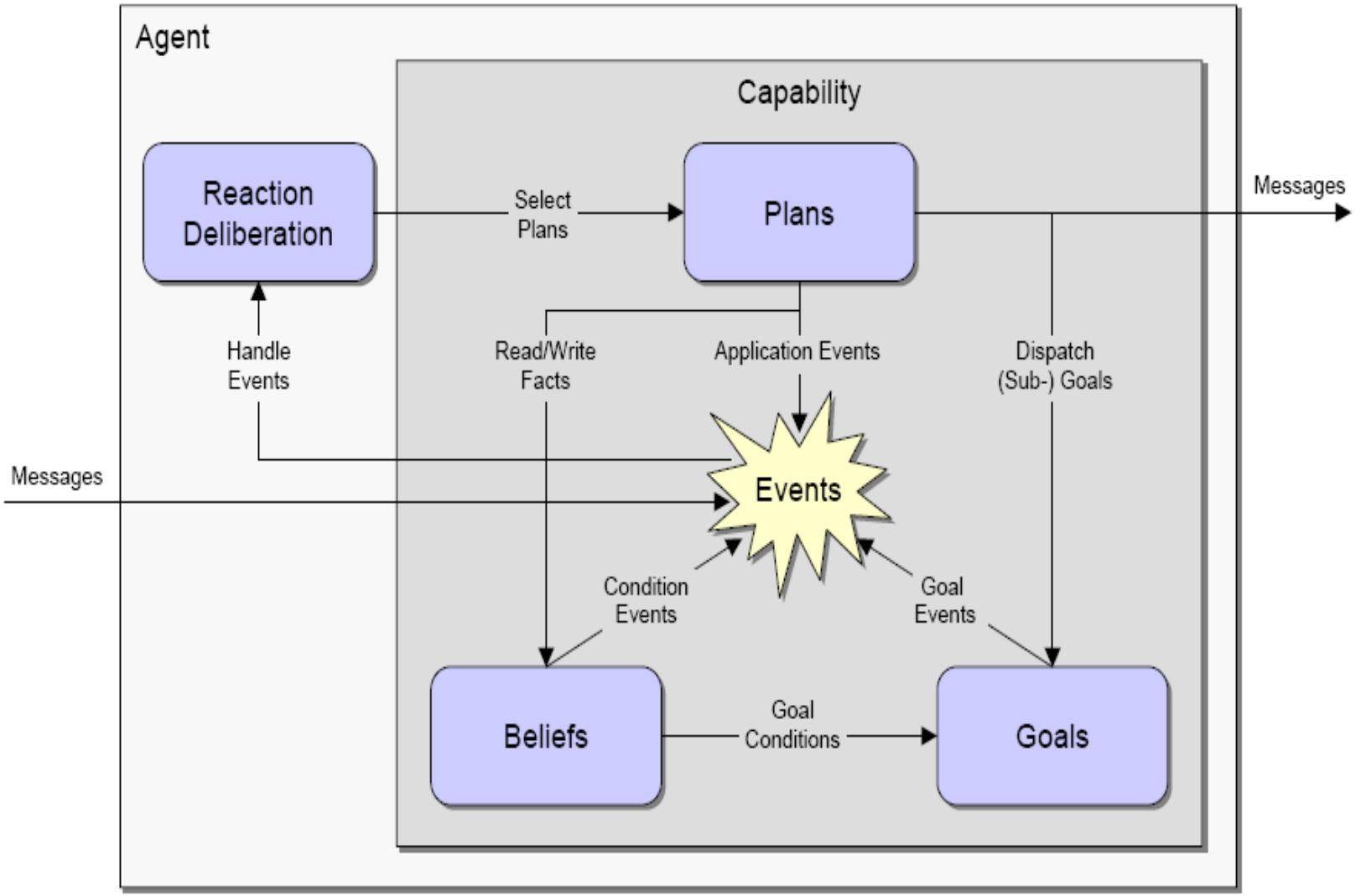
Jadex Abstract Agent Architecture



Plans *(intentions)*

- Represent *procedural knowledge*
  - Means for goal achievement and reacting to events
  - Agent has library of pre-defined plans
  - Interleaved stepwise execution
- Realisation of a plan
  - **Plan head** specified in ADF
  - **Plan body** coded in pure Java
- Assigning plans to goals/events
  - *Plan head* indicates ability to handle goals/events
  - Plan context / precondition further refines set of applicable plans

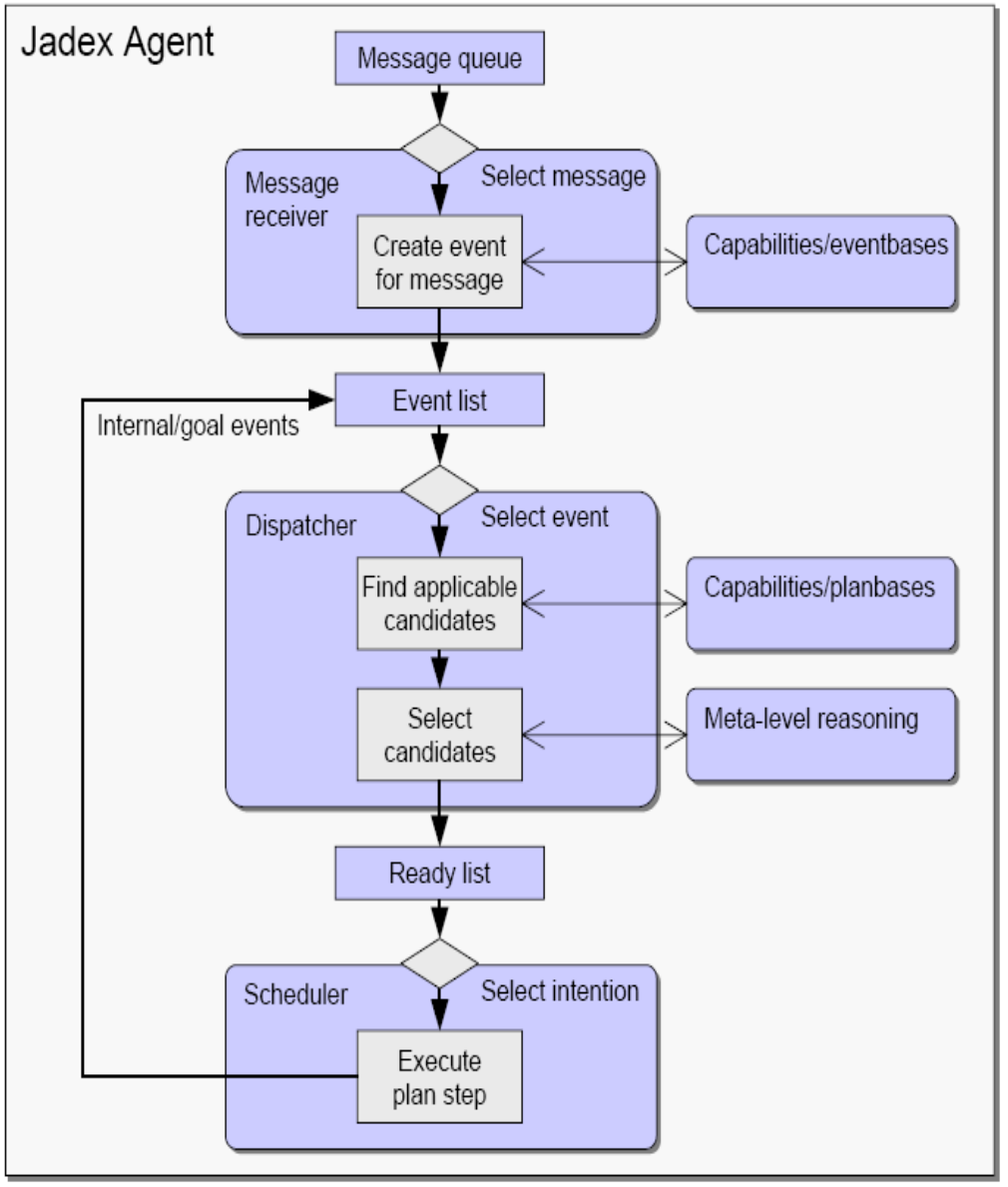
Jadex Abstract Agent Architecture



Events

- Three types of events
  - **Message** event denotes arrival/sending messages
  - **Goal** event denotes a new goal to be processed or that the state of an existing goal is changed
  - **Internal** event
    - **Timeout** event denotes that a timeout has occurred, e.g., waiting for arrival of messages/achieving goals/**`waitFor(duration)`** actions.
    - **Execute plan** event denotes plan to be executed without metalevel reasoning, e.g., plans with triggering condition
    - **Condition-triggered** event is generated when a state change occurs that satisfies the trigger of a condition

Jadex Event Dispatching Mechanism



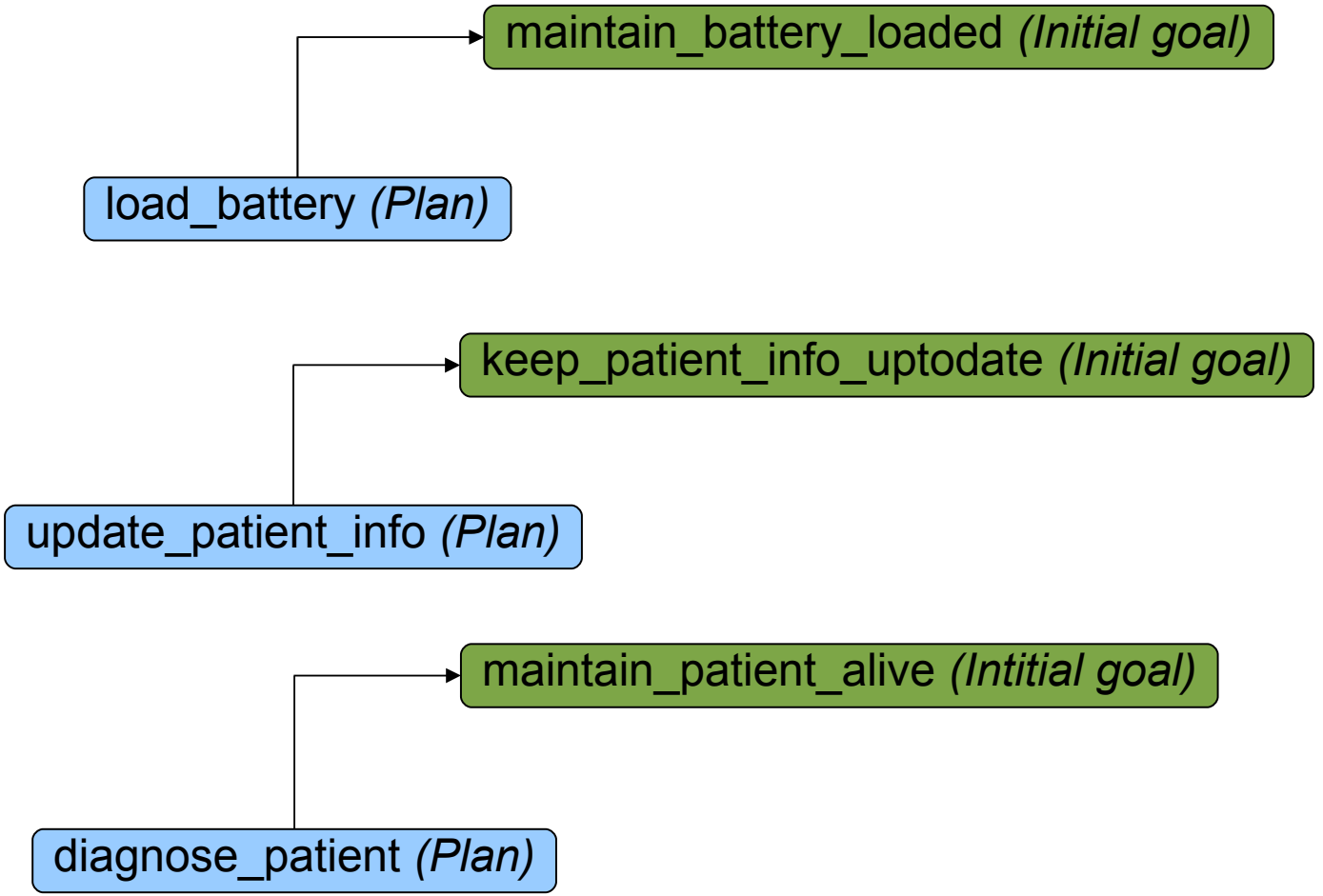
Overview

- Theoretical foundation of BDI
- Introduction to Jadex reasoning engine
- Developing tools in Jadex
- JADE example
- Implementation in Jadex
- Conslusions

Example in JADE

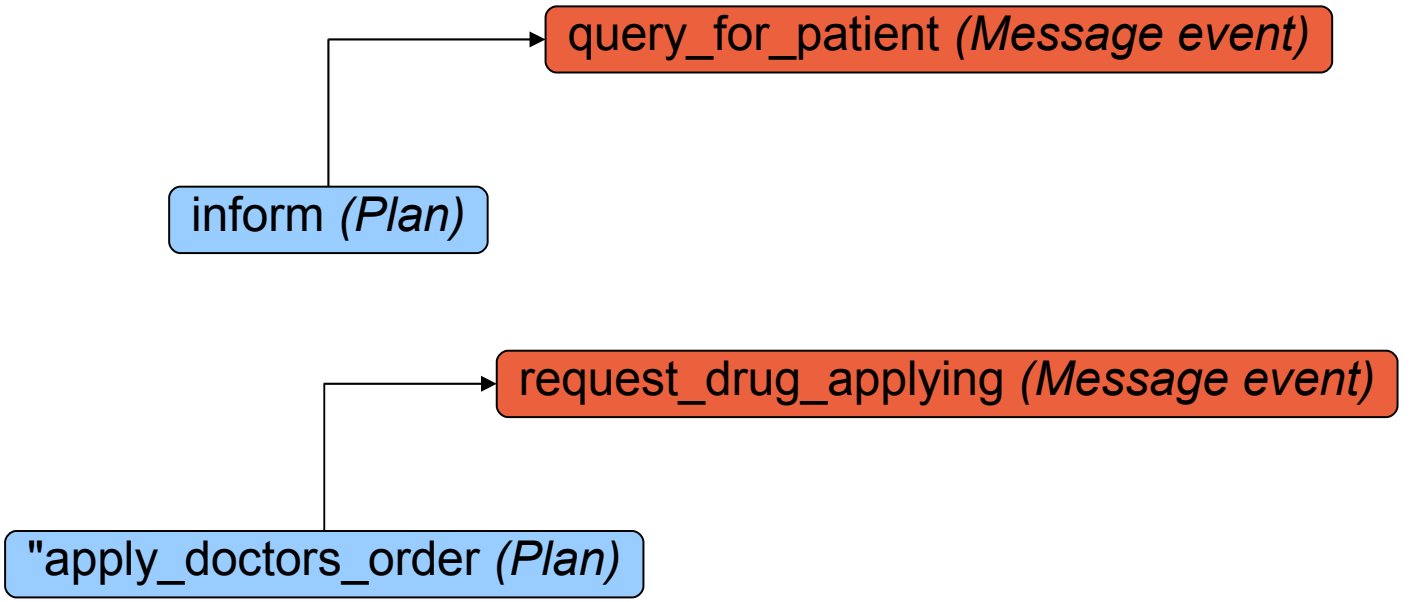
- Package **`ibspan.lab3.ex1`**
- Launching
  - **`bin\ex1-nurse.bat`**
  - **`bin\ex1-doc.bat`**
- Observation
  - *Patient*'s blood presure depends on her age, recently taken drugs and time flow
  - *Nurse* observes patient's blood pressure, informs *Doctor* about it and gives drugs on *Doctor*'s request
  - *Doctor* diagnoses *Patient*'s state and Doctor diagnoses Patient's state and orders the *Nurse* to keep *Patient*'s blood pressure at a specific average level

Doctor in BDI



- *Beliefs:* my_chargestate, patient_pressure, patient_is_alive, nurse

Nurse in BDI



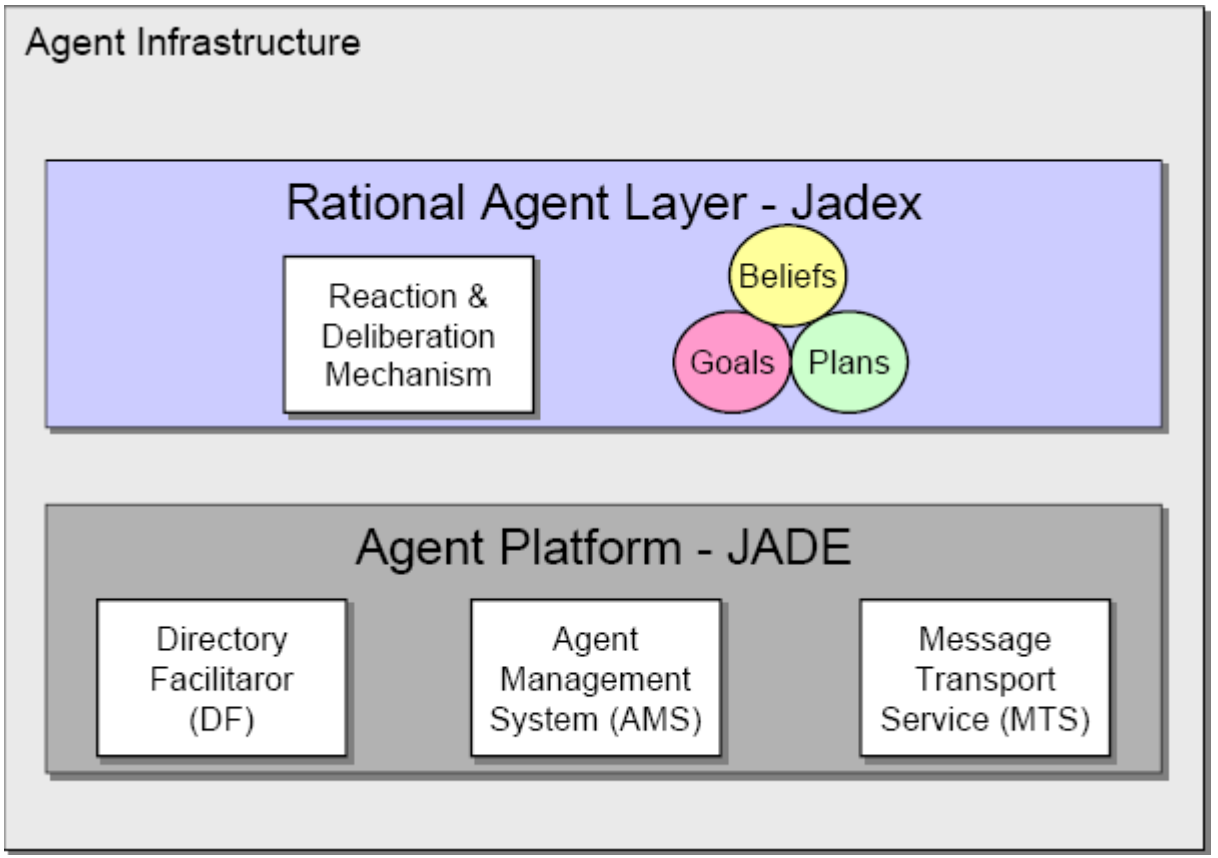
- *Beliefs:* my_patient, pressure, is_alive

Overview

- Theoretical foundation of BDI
- Introduction to Jadex reasoning engine
- Developing tools in Jadex
- JADE example
- Implementation in Jadex
- Conslusions

Platform adapters

- Is a BDI-extension (add-on) for the FIPA-compliant JADE multi-agent platform



Platform adapters

- Jadex is realized as pure reasoning engine.
- Can use any middleware platform providing services for agent managements and messaging
- Adapter required to access middleware platform
- Adapters realized for:
  - JADE
  - Standalone platform (from Jadex)

Jadex Standalone Adapter

- Fast and efficient execution environment
- Small memory footprint
- No support for mobility & persistence
- Contained in Jadex distribution (**`jadex_standalone.jar`**)
- Starting standalone platform
  `java jadex.adapter.standalone.Platform`

JADE Adapter

- Provides mobility & persitence
- Allows using standard JADE behaviours approach
- Not contained in the standard Jadex distribution
- Download & add to classpath:
  - from Jadex page:
    adapter (**`jadex_jadeadapter.jar`**)
  - from JADE page:
    official JADE jars (**`base64.jar, http.jar, iiop.jar,jade.jar, jadeTools.jar`**) and additionally Crimson (**`crimson.jar`**)
- Starting with JADE platform

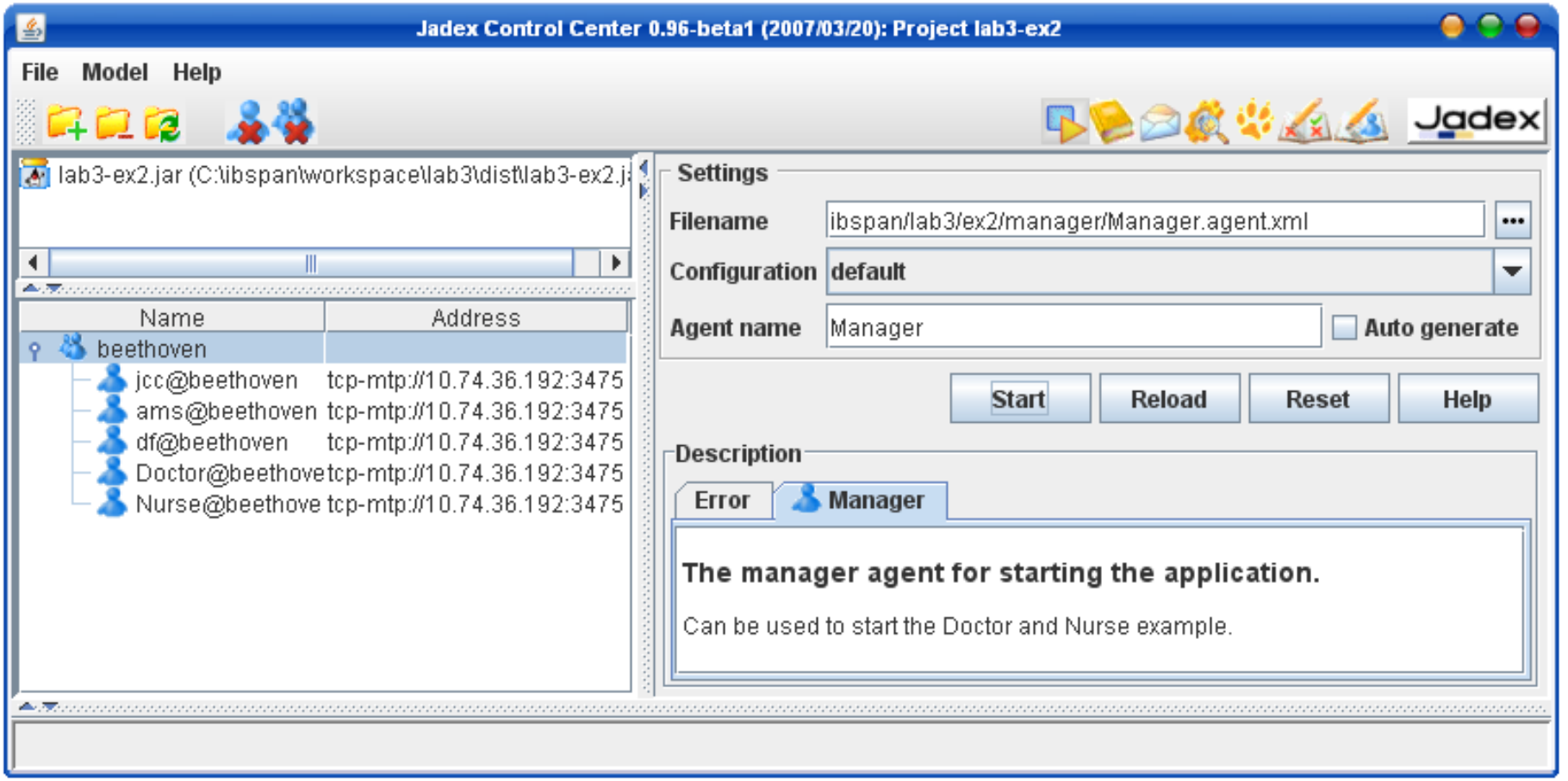
```
java jade.Boot
   rma:jadex.adapter.jade.tools.rma.rma
```

Jadex Control Center

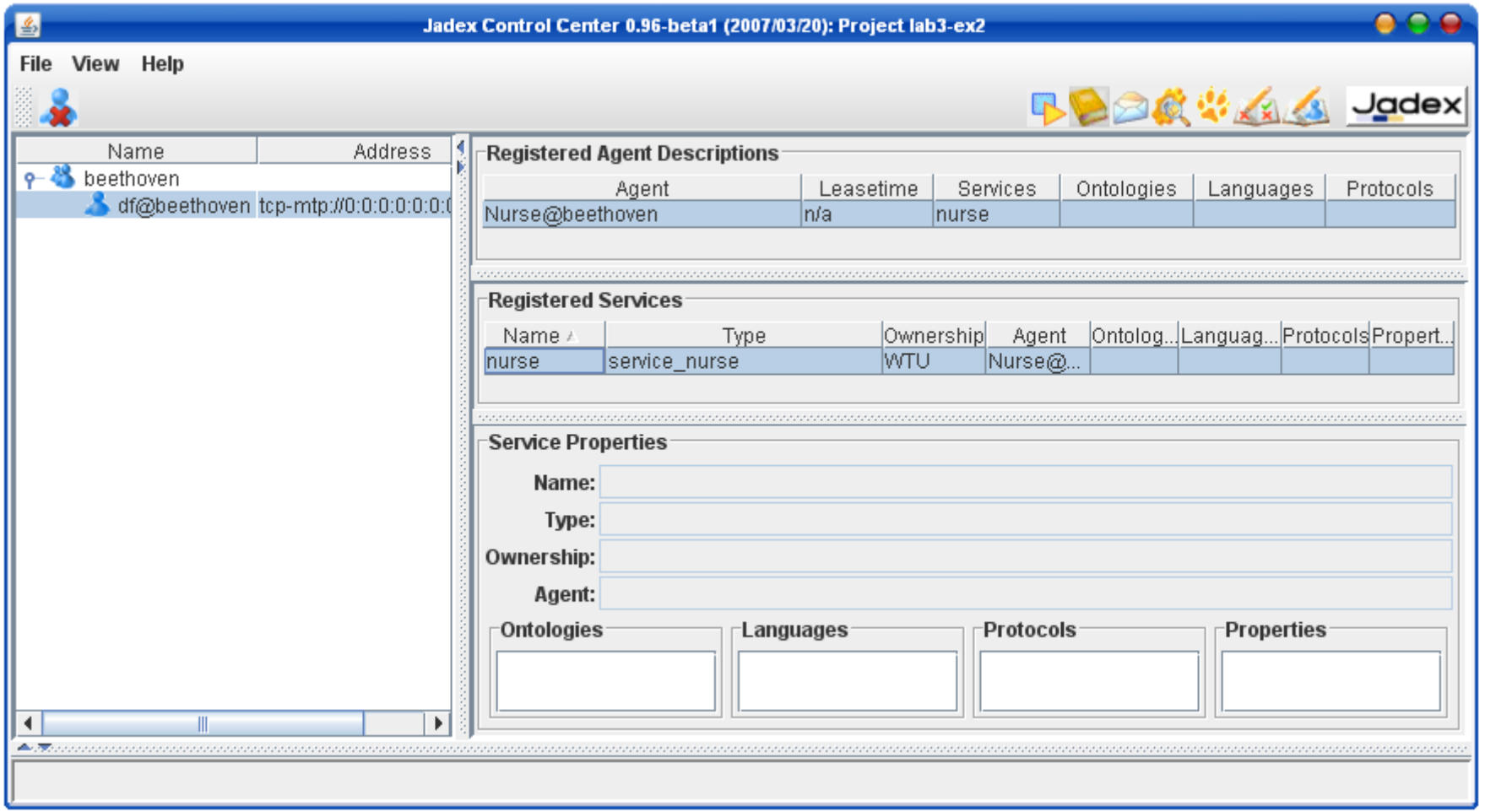
- Started per default when the Standalone platform is launched

- Provides:
  - project handling
  - central access point for all runtime toolset
  - functionalities provided by plug-ins in separate perspectives

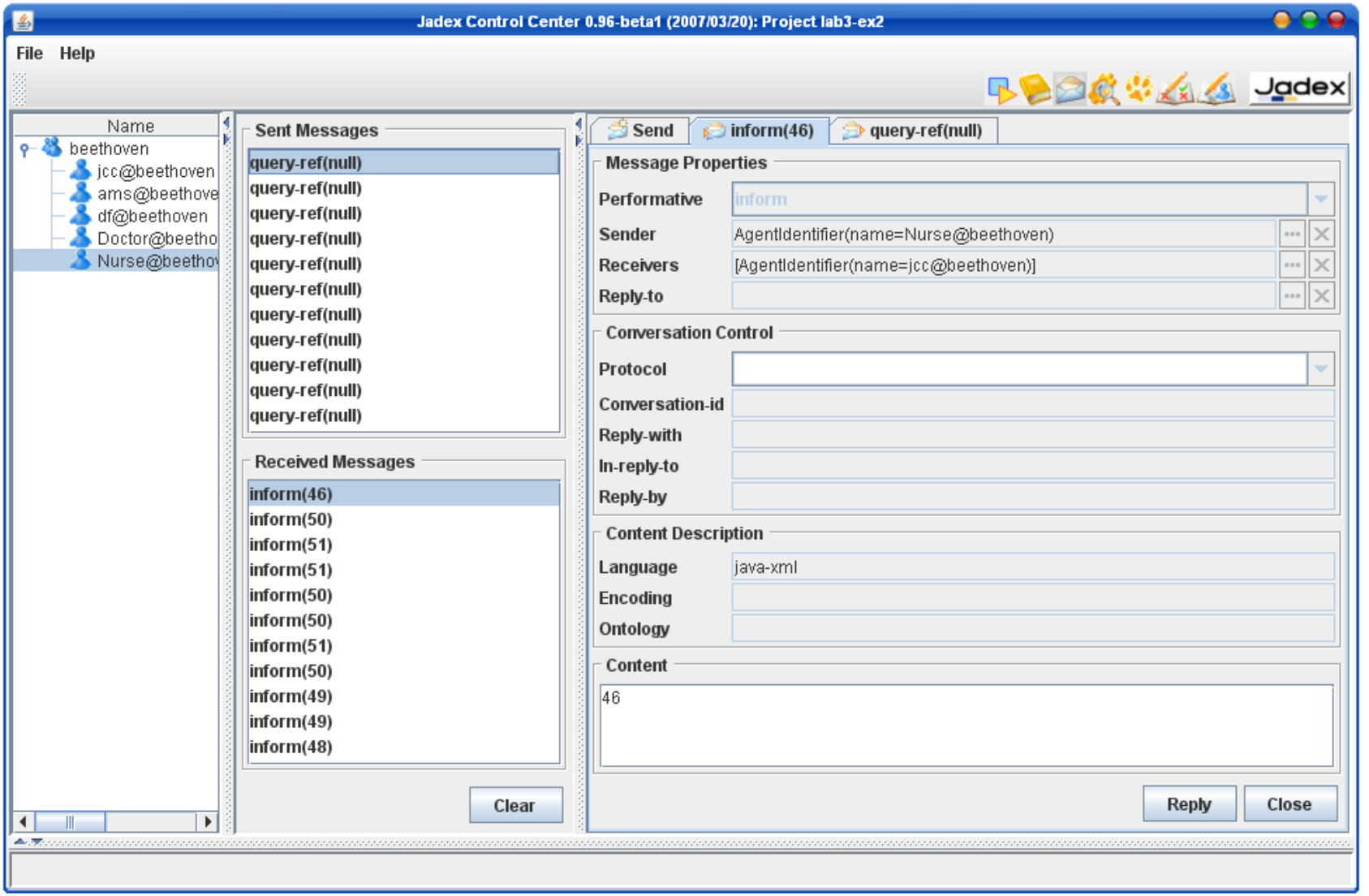
Jadex Control Center



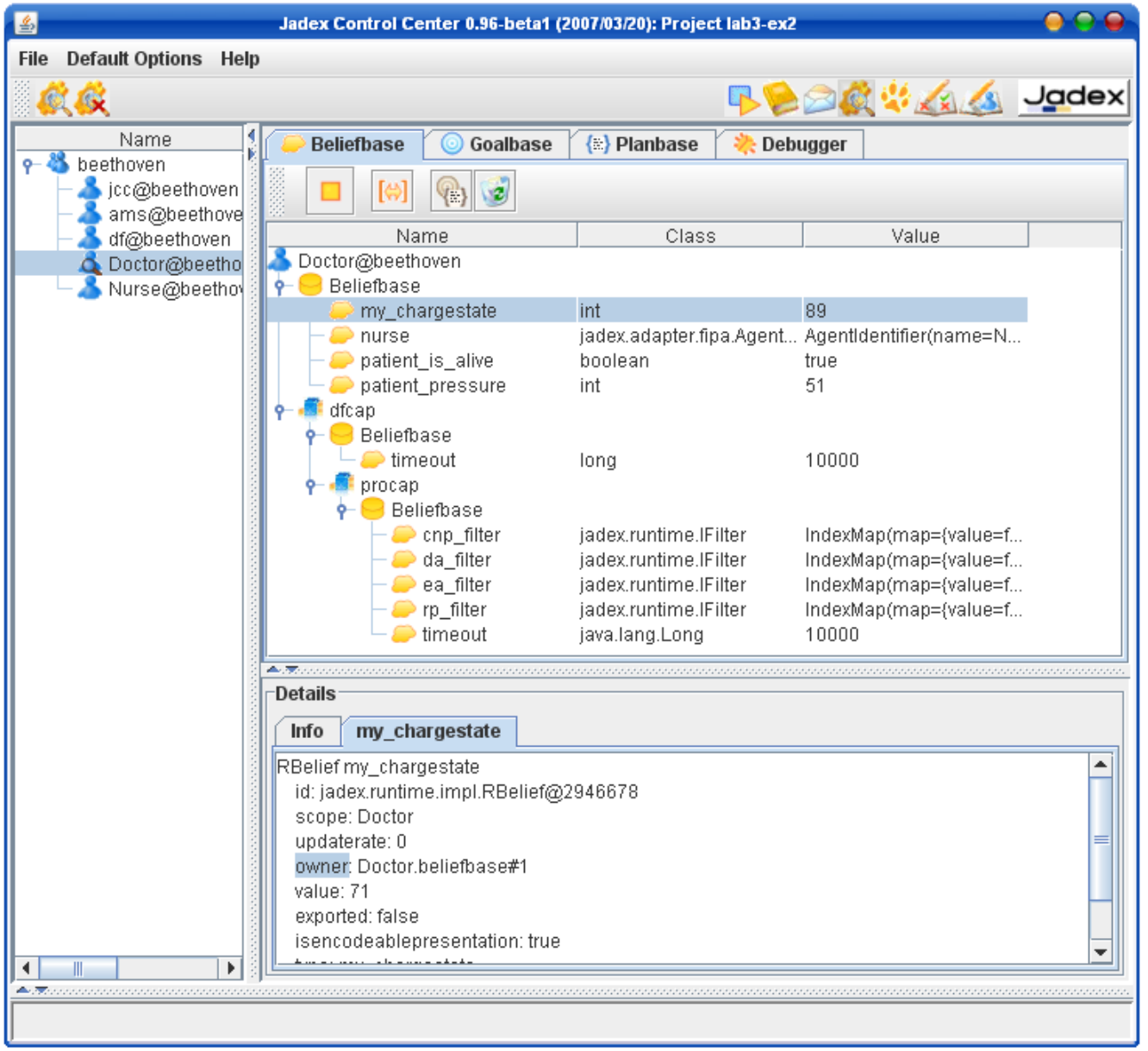
DF Browser


Jadex Control Center 0.96-beta1 (2007/03/20): Project lab3-ex2
File View Help
Jadex
Name
Address
beethoven
df@beethoven
tcp-mtp://0:0:0:0:0:0:0:
Registered Agent Descriptions
Agent
Leasetime
Services
Ontologies
Languages
Protocols
Nurse@beethoven
n/a
nurse
Registered Services
Name
Type
Ownership
Agent
Ontolog...
Languag...
Protocols
Propert...
nurse
service_nurse
WTU
Nurse@...
Service Properties
Name:
Type:
Ownership:
Agent:
Ontologies
Languages
Protocols
Properties

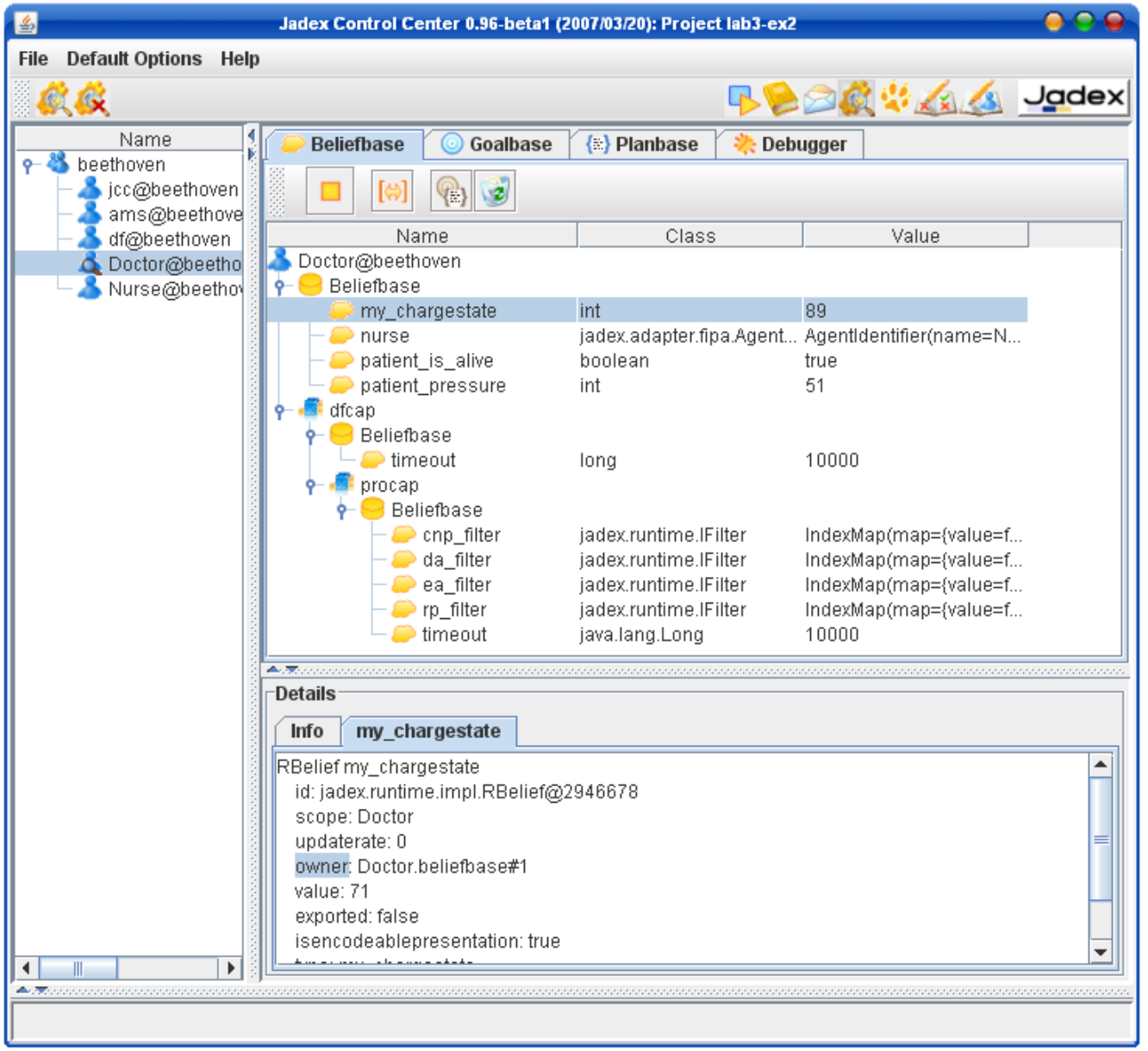
Conversation Center



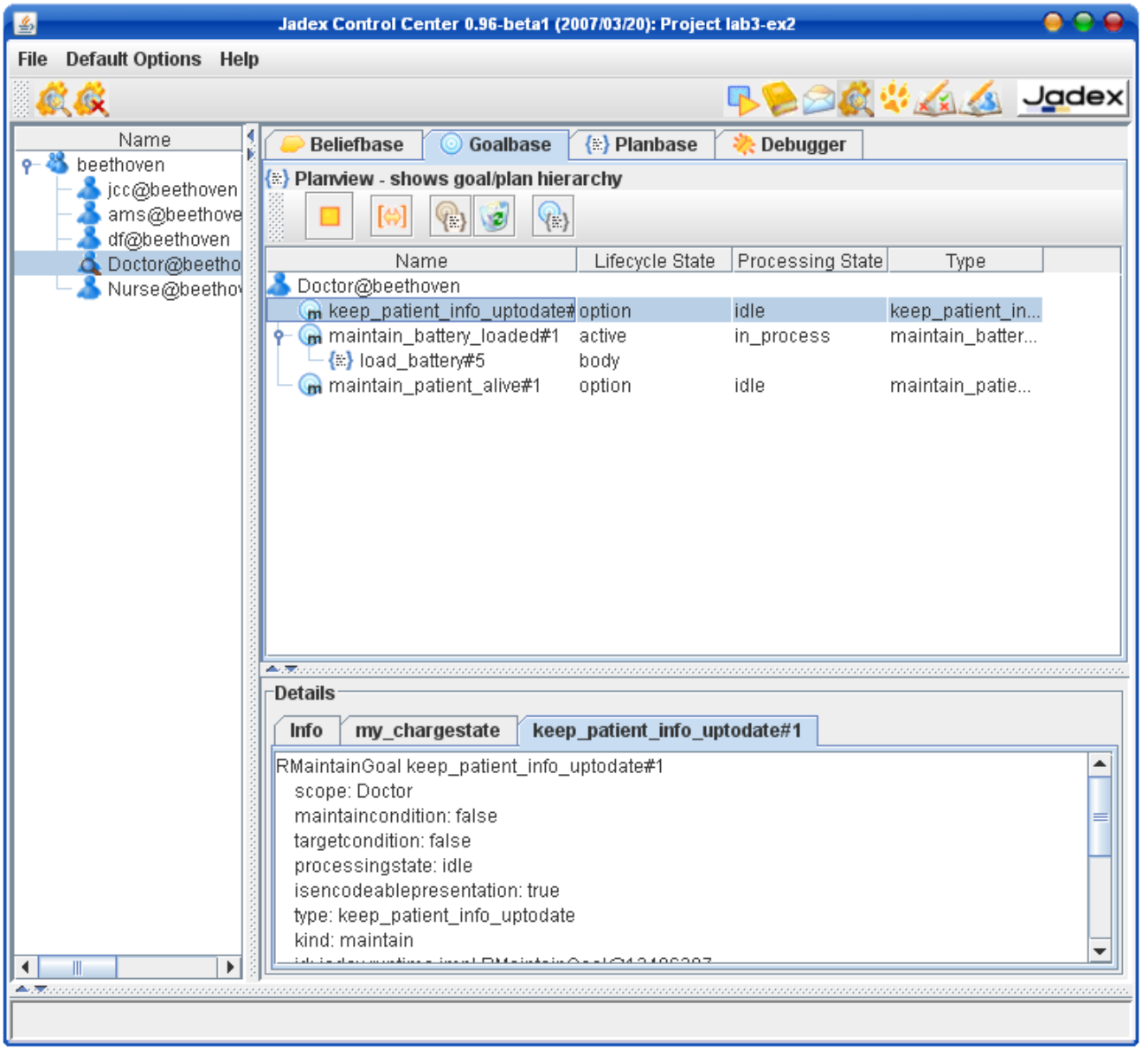
Introspector



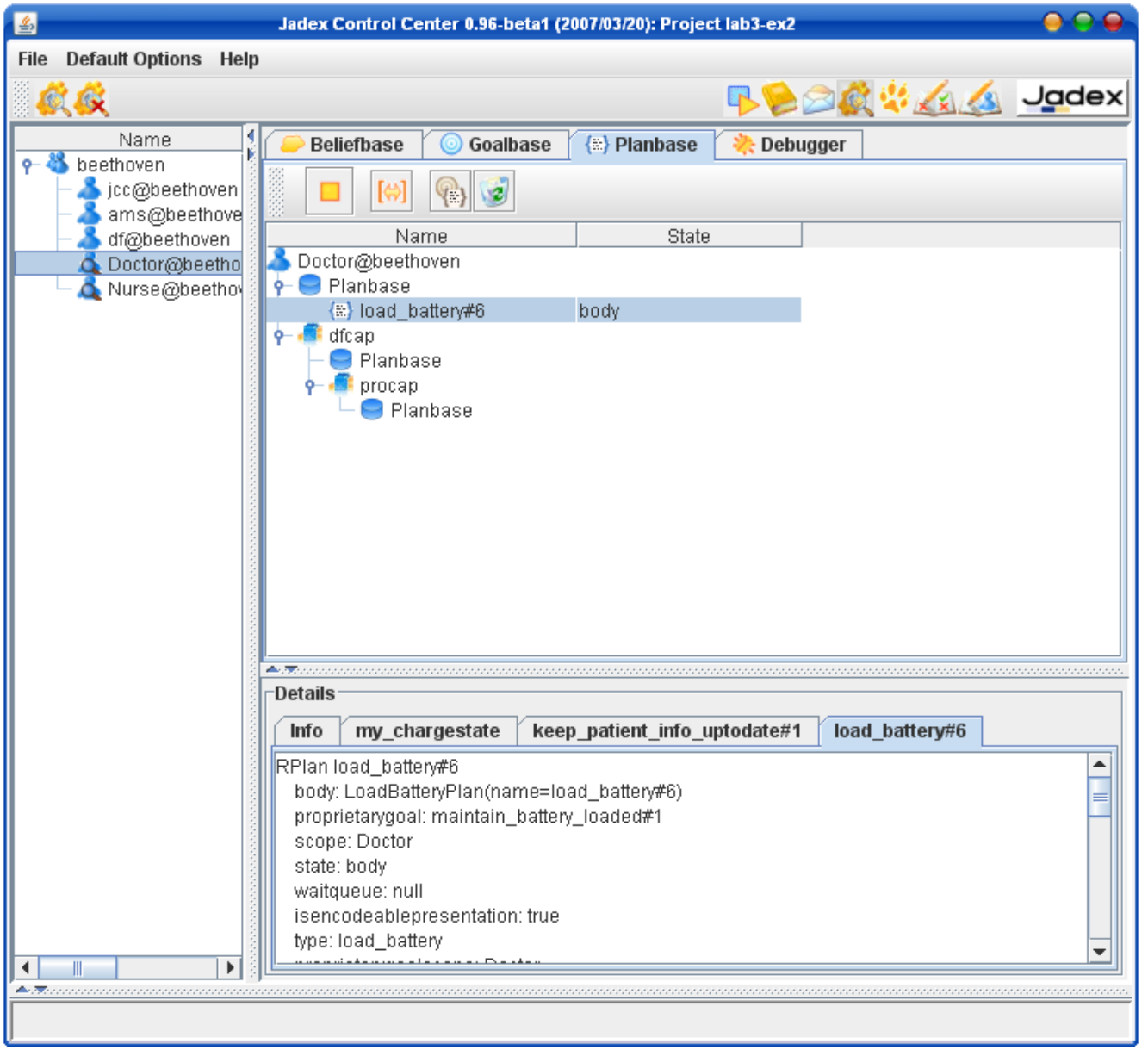
Introspector



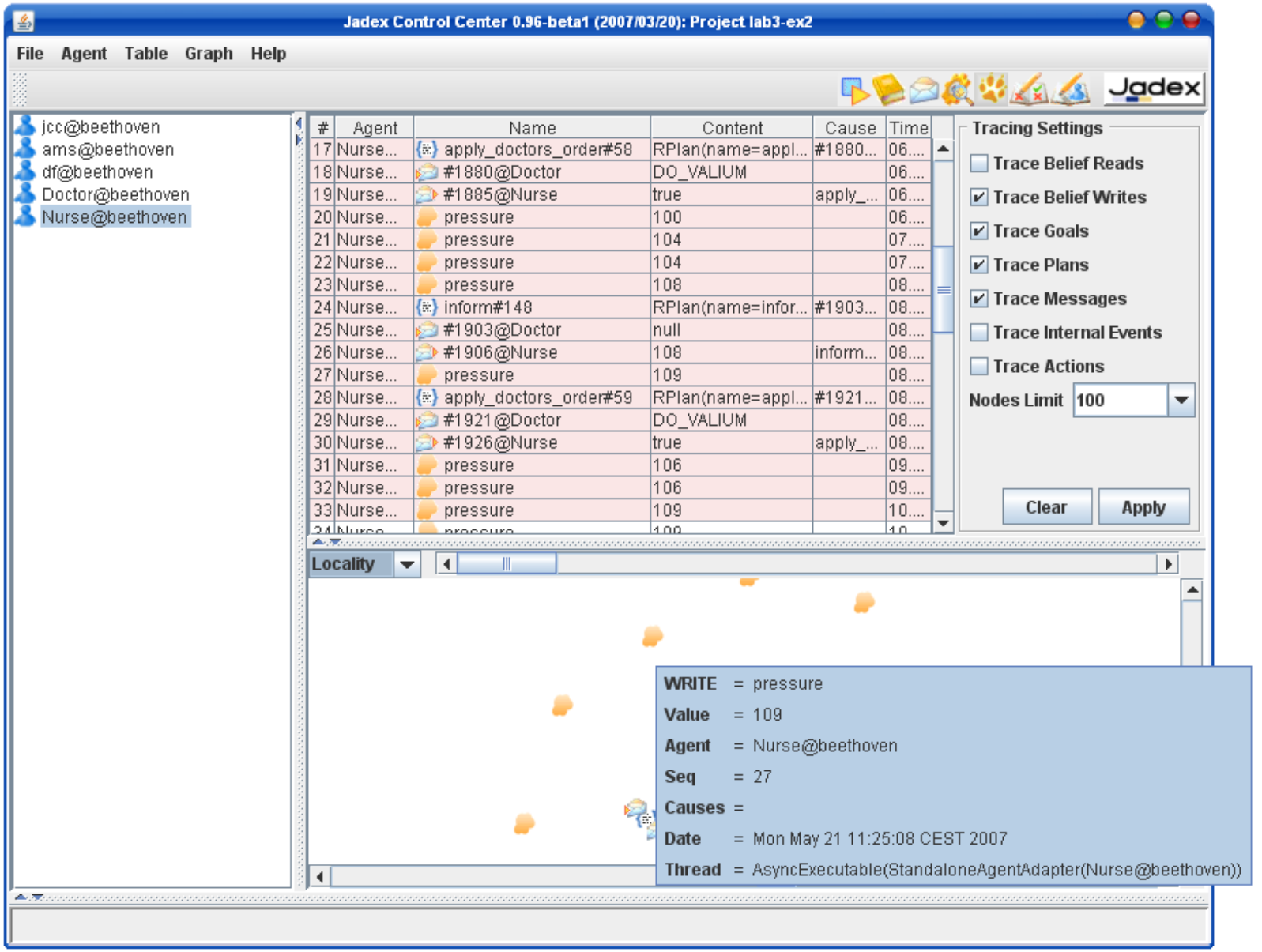
Introspector



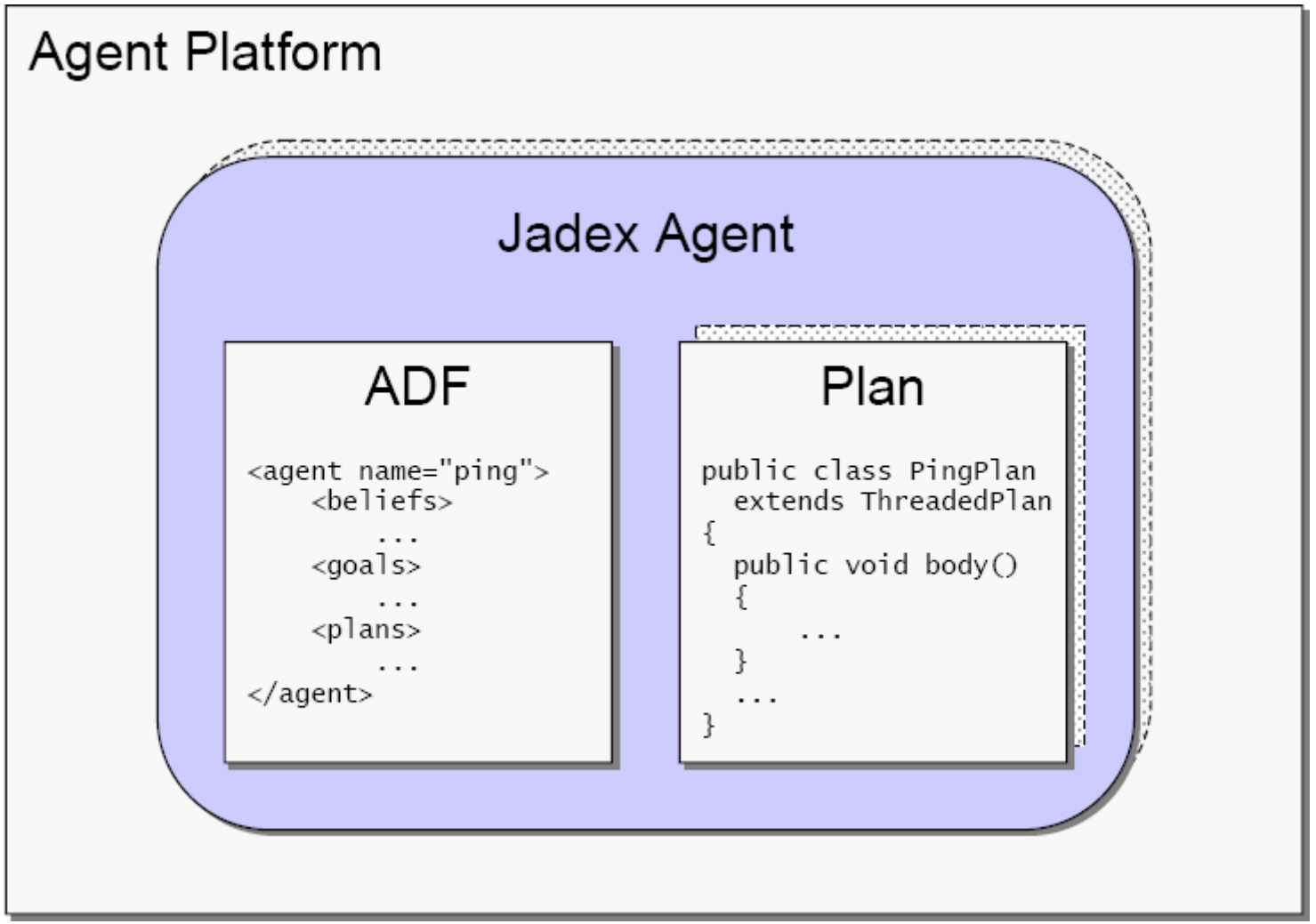
Introspector



Tracer


Jadex Control Center 0.96-beta1 (2007/03/20): Project lab3-ex2
File Agent Table Graph Help
Jadex
jcc@beethoven
ams@beethoven
df@beethoven
Doctor@beethoven
Nurse@beethoven
Agent Name Content Cause Time
17 Nurse... apply_doctors_order#58 RPlan(name=appl... #1880... 06....
18 Nurse... #1880@Doctor DO_VALIUM 06....
19 Nurse... #1885@Nurse true apply_... 06....
20 Nurse... pressure 100 06....
21 Nurse... pressure 104 07....
22 Nurse... pressure 104 07....
23 Nurse... pressure 108 08....
24 Nurse... inform#148 RPlan(name=infor... #1903... 08....
25 Nurse... #1903@Doctor null 08....
26 Nurse... #1906@Nurse 108 inform... 08....
27 Nurse... pressure 109 08....
28 Nurse... apply_doctors_order#59 RPlan(name=appl... #1921... 08....
29 Nurse... #1921@Doctor DO_VALIUM 08....
30 Nurse... #1926@Nurse true apply_... 08....
31 Nurse... pressure 106 09....
32 Nurse... pressure 106 09....
33 Nurse... pressure 109 10....
Tracing Settings
Trace Belief Reads
Trace Belief Writes
Trace Goals
Trace Plans
Trace Messages
Trace Internal Events
Trace Actions
Nodes Limit 100
Clear
Apply
Locality
WRITE = pressure
Value = 109
Agent = Nurse@beethoven
Seq = 27
Causes =
Date = Mon May 21 11:25:08 CEST 2007
Thread = AsyncExecutable(StandaloneAgentAdapter(Nurse@beethoven))

Components of a Jadex Agent

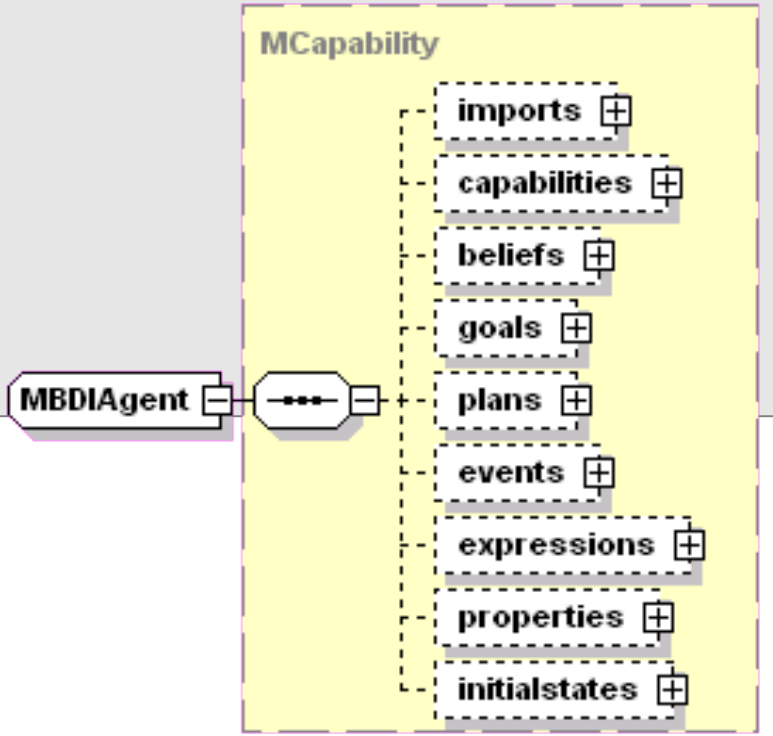


Agent Definition File (ADF)

- ADF defines agent startup properties:
  - initial goals and beliefs
  - heads of plans
- ADF syntax and semantics:
  - ADF is written in XML
  - semantics defined by XML schema: which elements can be specified inside an agent definition file
  - XML schema defined in **`jadex/docs/schema/jadex-0.95.html`**

Agent Definition File (ADF)

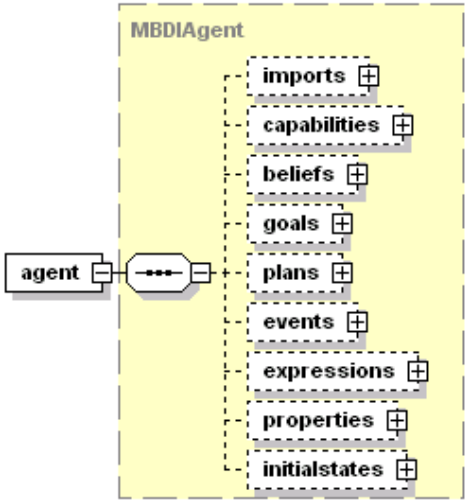
```
<agent xmlns="http://jadex.sourceforge.net/jadex"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jadex.sourceforge.net/jadex
                        http://jadex.sourceforge.net/jadex-0.95.xsd"
    name="..." package="...">

    <imports>...</imports>
    <capabilities>...</capabilities>
    <beliefs>...</beliefs>
    <goals>...</goals>
    <plans>...</plans>
    <events>...</events>
    <expressions></expressions>
    <properties>...</properties>
    <initialstates>...</initialstates>

</agent>
```



Agent Definition File (ADF)

element **agent**

diagram



namespace: http://jadex.sourceforge.net/jadex

type: extension of **MBDIAgent**

children: **imports capabilities beliefs goals plans events expressions properties initialstates**

attributes

Name	Type	Use	Default	Fixed	Annotation
name	xs:string				documentation The elements name.
description	xs:string	optional			documentation The elements optional description text.
package	xs:string	optional			documentation The package to which this capability belongs.
abstract	xs:boolean	optional	false		documentation When a capability is declared as abstract, it cannot be used directly for execution. Instead there need to be some implementation for this capability that will be resolved from the capability identifier.
propertyfile	xs:string	optional	jadex.config.runtime		

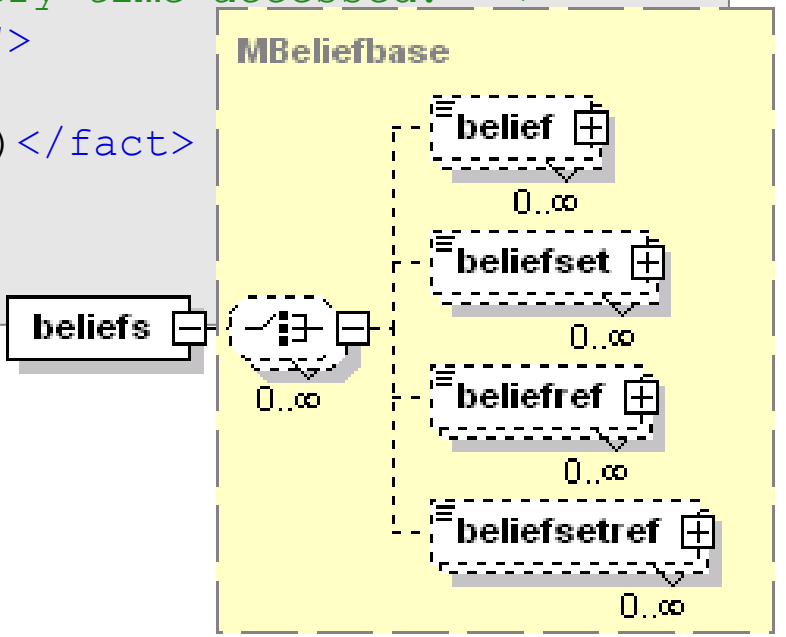
annotation: documentation Defines a new agent type.

Agent Definition File (ADF)

- When an ADF is loaded:
  - Java objects are created for the XML elements defined in the ADF, e.g.
    - *belief* → **`jadex.model.IMBelief`**
    - *goal* → **`jadex.model.IMGoal`**
    - *plan* → **`jadex.model.IMPlan`**

Beliefs

```
<beliefs>
    <!-- The patient (of age of 90), this Nurse takes care about. -->
    <belief name="my_patient" class="Patient">
        <fact>new Patient(90)</fact>
    </belief>

    <!-- Patient's blood pressure updated every 0.5 second. -->
    <belief name="pressure" class="int" updaterate="500">
        <fact>$beliefbase.my_patient.getPressure()</fact>
    </belief>

    <!-- Is patient alive flag, updated every time accessed. -->
    <belief name="is_alive" class="boolean">
        <fact evaluationmode="dynamic">
            $beliefbase.my_patient.isAlive()</fact>
    </belief>

</beliefs>
```



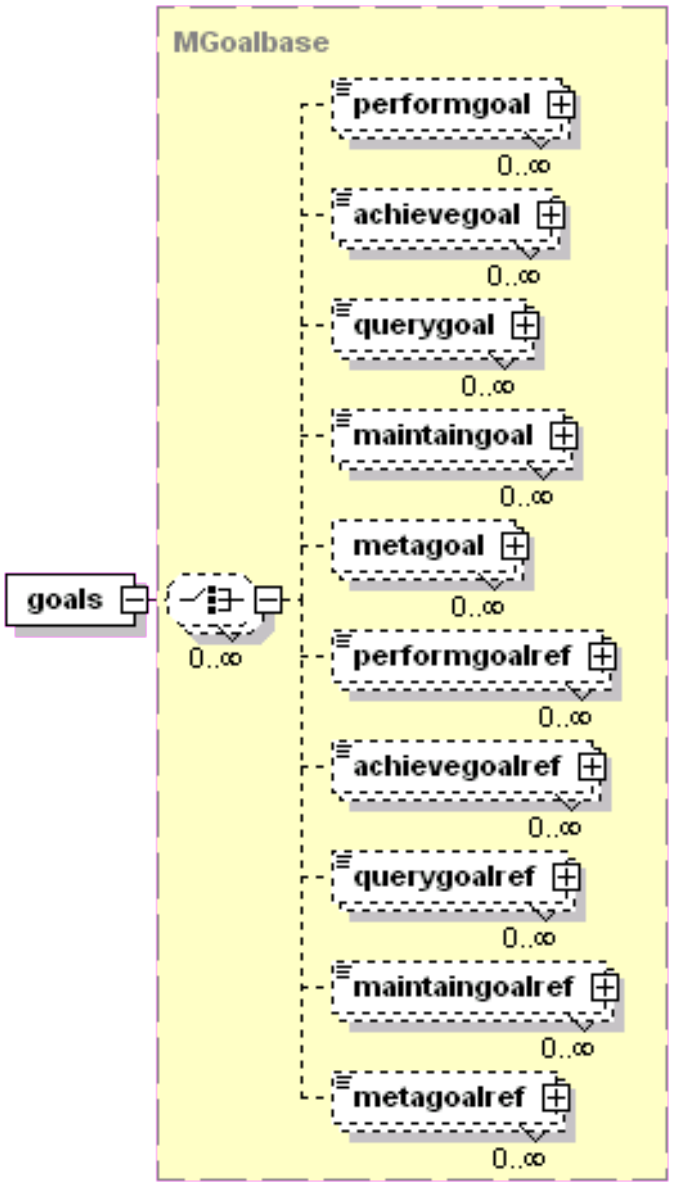
Access to Beliefs from Plans

- Methods:
  - **`getFact()`** – get the fact of a belief
  - **`setFact(Object fact)`** – set a fact of a belief
  - **`isAccessible()`** – is this belief accessible
- Example:

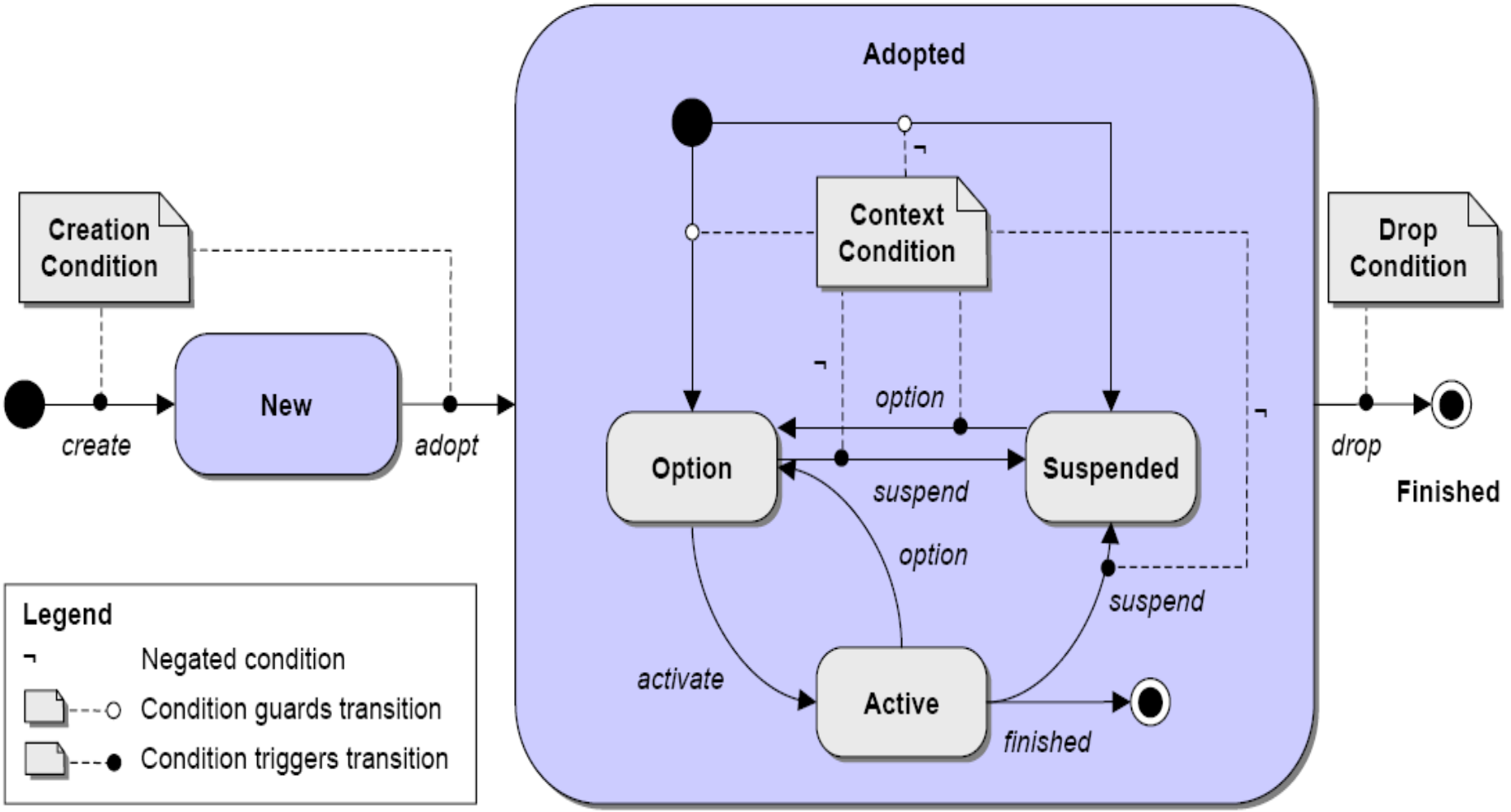
```
Integer pressure = (Integer) getBeliefbase().getBelief("pressure")
                .getFact();
```

```
// Updating information about patient consumes some energy...
int charge = (Integer)
  getBeliefbase().getBelief("my_chargestate").getFact();
getBeliefbase().getBelief("my_chargestate").setFact(
  new Integer(charge - 2));
```

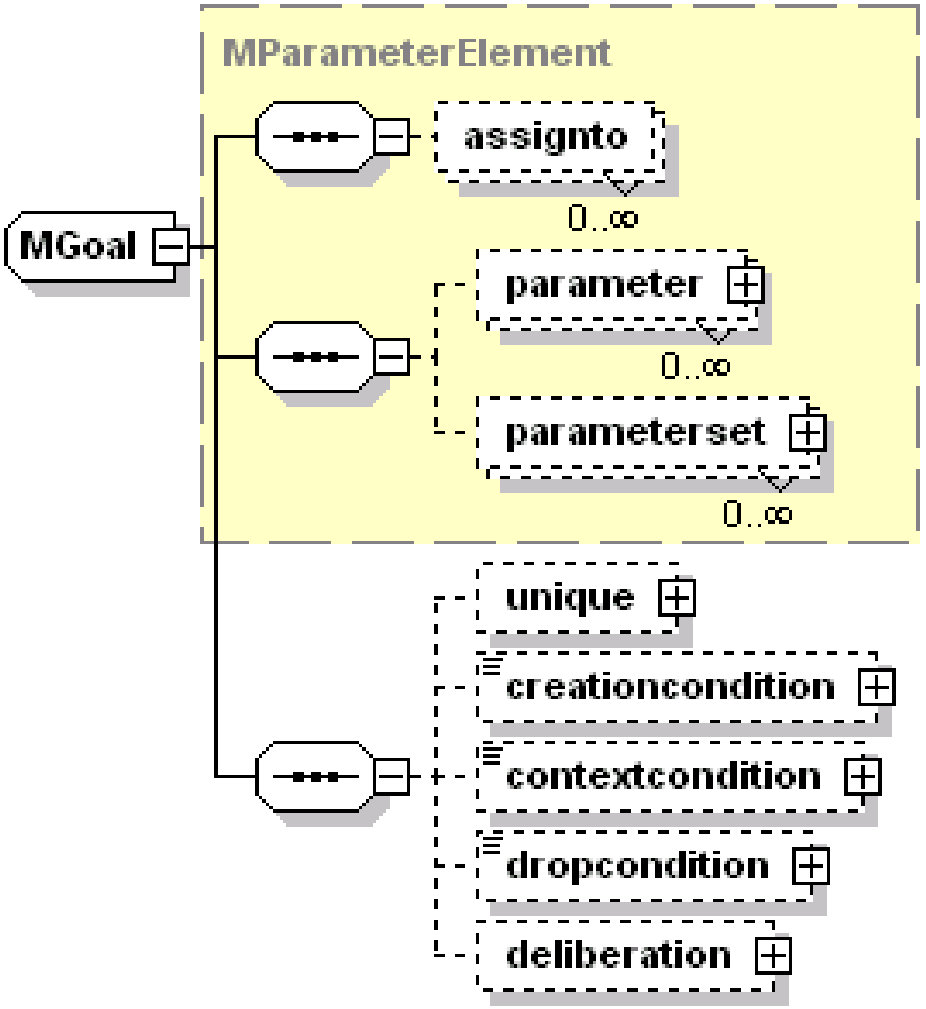
Goals


MGoalbase
goals
0..∞
performgoal
0..∞
achievegoal
0..∞
querygoal
0..∞
maintaingoal
0..∞
metagoal
0..∞
performgoalref
0..∞
achievegoalref
0..∞
querygoalref
0..∞
maintaingoalref
0..∞
metagoalref
0..∞

Goal Lifecycle



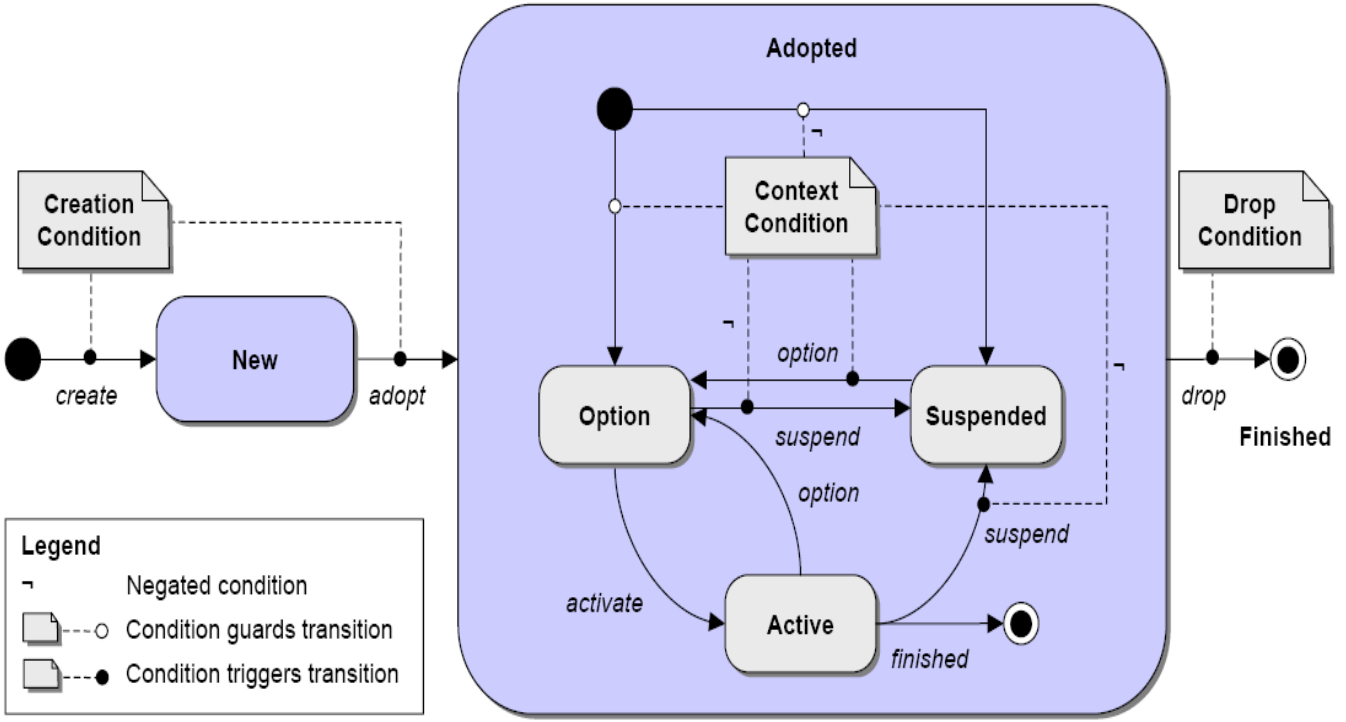
Goal


MParameterElement
MGoal
assignto
0..∞
parameter
0..∞
parameterset
0..∞
unique
creationcondition
contextcondition
dropcondition
deliberation

Goal Creation

- **<initialgoal>**: Initial goals are created and adopted as top-level goals when an agent is born
- **<creationcondition>**: When the creation condition triggers, then one or more goal instances are created and adopted as top-level goal(s).
- Plans may directly create goals and dispatch them as subgoals. These goals are adopted as subgoals of the plan's root goal. When a plan terminates or is aborted, all not yet finished subgoals are aborted automatically.
- Plans may also create goals and dispatch them as top-level goals. Once adopted, such a goal exists independently of the plan that created it.

Goal Lifecycle



- **<contextcondition>**: indicates when Active/Option goal should be suspended
- **<dropcondition>**: indicates when adopted goals should be dropped
- **<deliberation>**: indicates which Option goals should be (de)activated (*inhibition* and *cardinality*)

Goal Flags

- **`retry`** *{**`true`**, **`false`**}*:
  - the goal should be retried or redone, until it is reached, or no more plans are available, which can handle the goal.
  - Default=**`true`**
- **`exclude`** *{**`when_tried, when_succeeded, when_failed, never`**}*:
  - used in conjunction with **`retry`**; when retrying a goal, only plans should be called, that where not already executed for that goal.
  - Default=**`when_tried`**
- **`posttoall`** *{**`true, false`**}*:
  - enables *parallel* processing of a goal, by dispatching the goal to all applicable plans at once.
  - Default=**`false`**

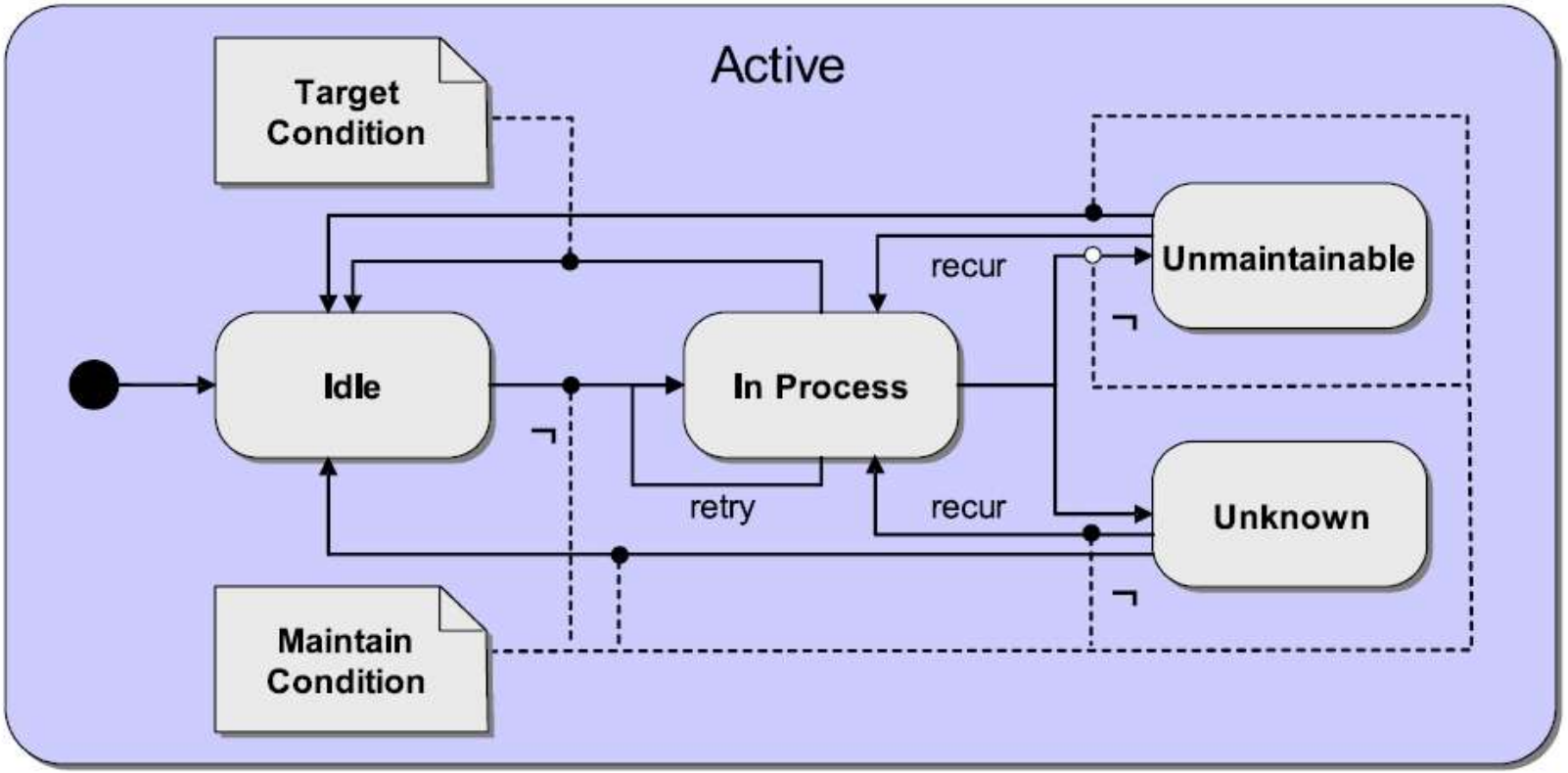
Goal Flags

- **`retrydelay`** *(positive long value)*
  - optional waiting time (in milliseconds)
    - Without retrydelay goal processing works as follows:
      goal → *plan 1* → *plan 2* → *plan 3* → *...*
      until the goal is failed or succeeded.
    - The retrydelay just specifies a delay in milliseconds before trying the next plan, when the previous plan has finished, i.e.:
      goal → *plan 1* → *wait* → *plan 2* → *wait* → *plan 3* → *...*
      until goal fails or succeeds.
    - This is e.g. useful, when already tried plans are not excluded from the applicable plan set, leading to the same plan being tried over and over again.
  - Default=`0`

Goal Flags

```
!-- Maintain correct patient's blood pressure, but only if Doctor
          has energy. -->
<maintaingoal name="maintain_patient_alive" exclude="never"
retry="true" retrydelay="2500">
    <contextcondition>$beliefbase.my_chargestate >
      0</contextcondition>
    <!-- Engage in actions when the pressure is out of [50,100] range.
      -->
    <maintaincondition> $beliefbase.patient_pressure >= 50 &amp;&amp;
       $beliefbase.patient_pressure &lt;= 100
    </maintaincondition>
</maintaingoal>
```

Maintain Goals



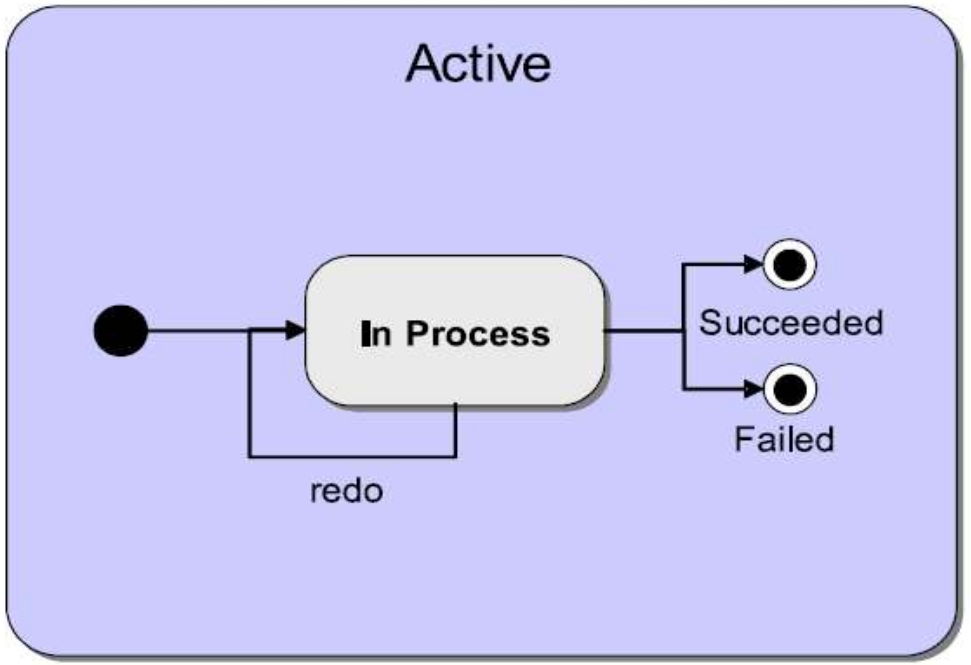
Maintain Goals

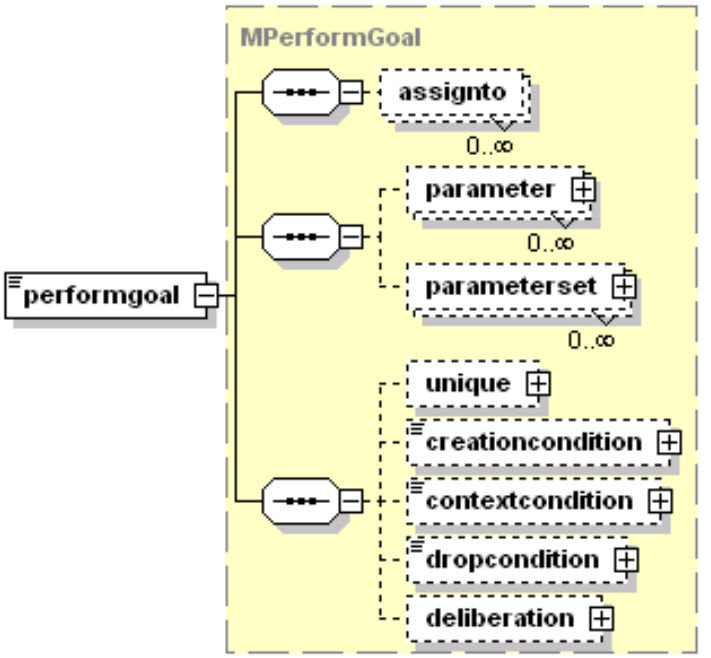
- Keep-Operational (keep track of the battery state and charge it when necessary)

```
<!-- Observe the battery state. -->
<maintaingoal name="maintain_battery_loaded" exclude="never"
    retry="true">
  ...
    <!-- Engage in actions when the state is below 20. -->
    <maintaincondition> $beliefbase.my_chargestate >= 20
    </maintaincondition>
    <!-- The goal is satisfied when the charge state is 100. -->
    <targetcondition> $beliefbase.my_chargestate >= 100
    </targetcondition>
</maintaingoal>
```

- To avoid the agent loading only until 21% (which satisfies the maintain condition), the extra **<targetcondition>** is used. It ensures that the agent stays loading until the battery is fully recharged.

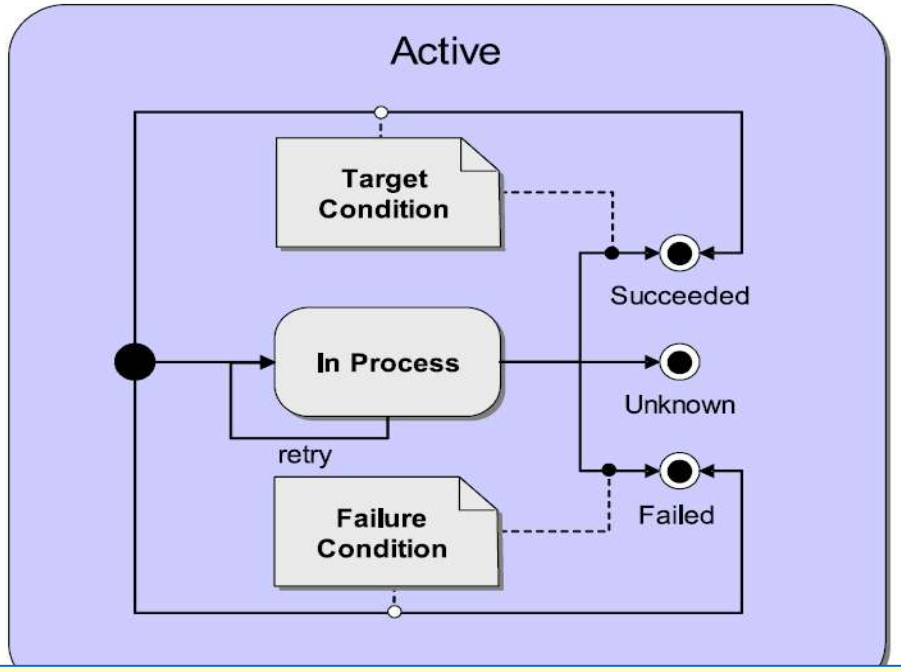
Perform Goals

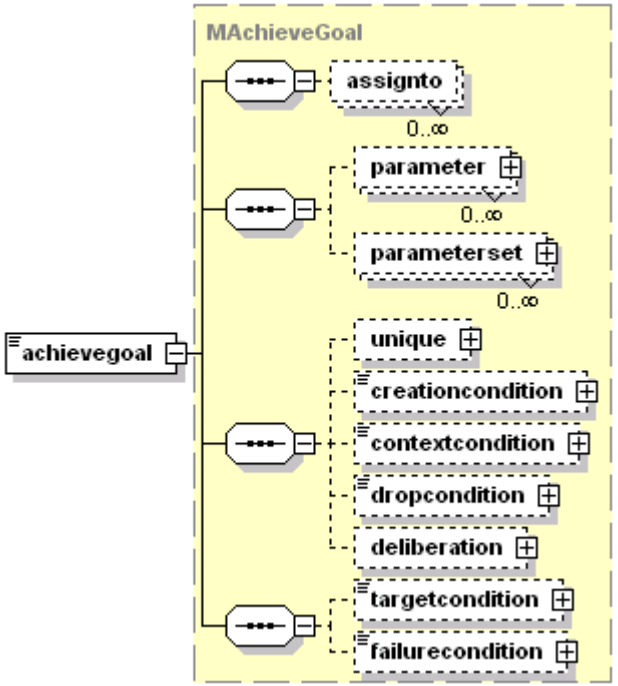





```
<!-- Look out for waste when nothing better to do, what means that
     the agent is not cleaning, not loading and it is daytime. -->
<performgoal name="performlookforwaste" retry="true" exclude="never">
  <contextcondition>
    $beliefbase.daytime
  </contextcondition>
</performgoal>
```

Achieve Goals

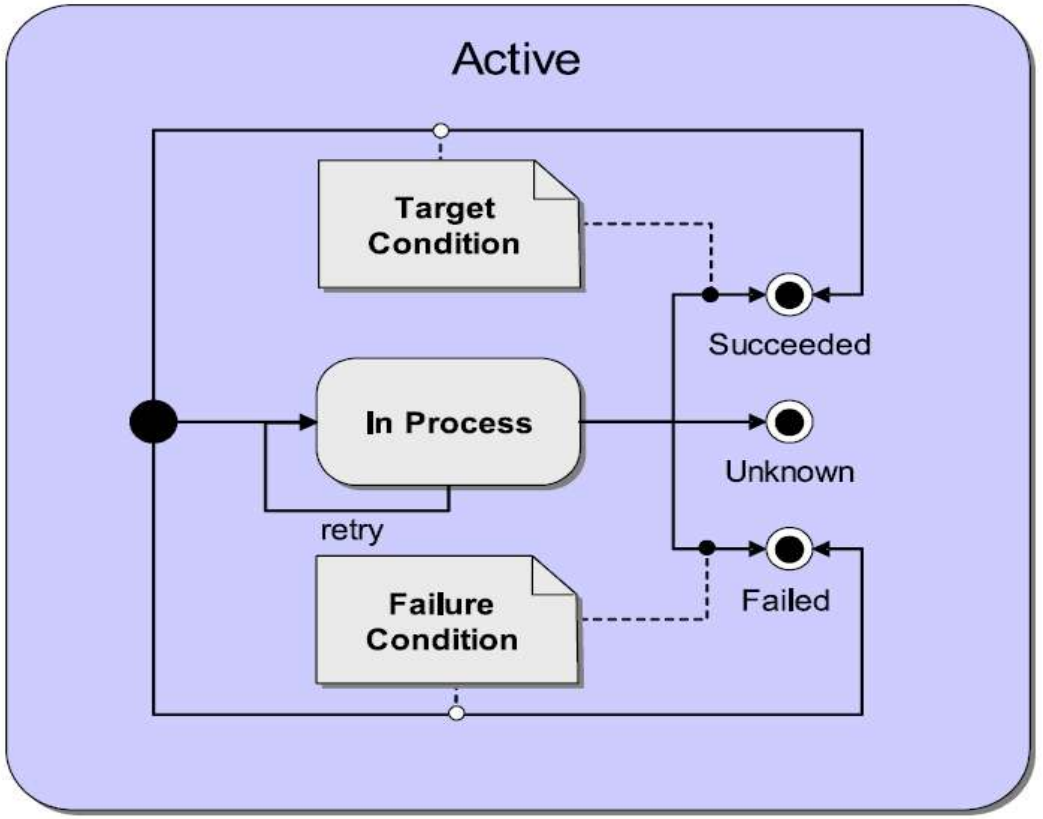


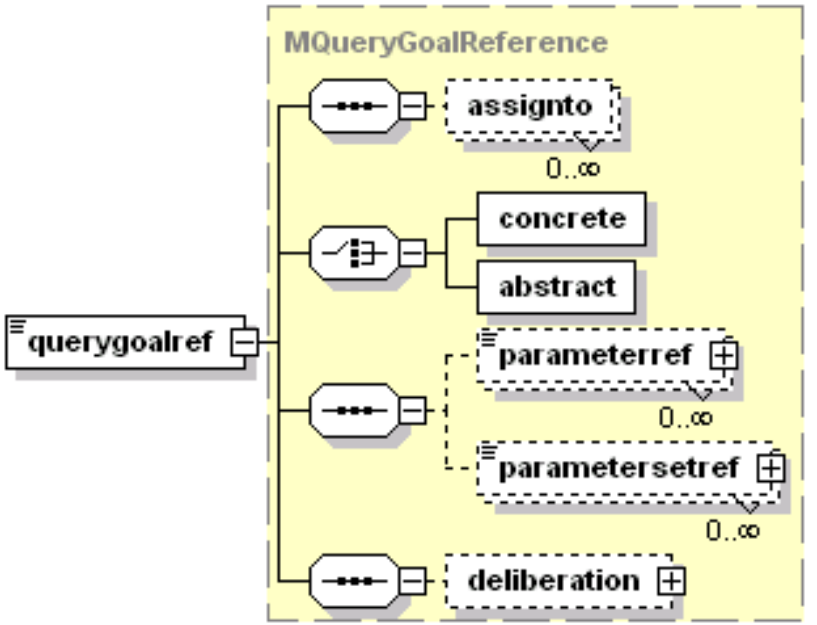



```
<!-- Drop a piece of waste into a wastebin. -->
<achievegoal name="achievedropwaste" retry="true" exclude="never">
  <parameter name="wastebin" class="Wastebin"/>
  <!-- The goal has failed when the aimed wastebin is full. -->
  <failurecondition>
    (select one Wastebin $wastebin
    from $beliefbase.wastebins
    where $goal.wastebin.getId().equals($wastebin.getId())).isFull()
  </failurecondition>
</achievegoal>
```

```
<!-- Try to move to the specified location. -->
<achievegoal name="achievemoveto">
  <parameter name="location" class="Location"/>
  <!-- The goal has been reached when the agent's location is
       near the target position as specified in the parameter. -->
  <targetcondition>
    $beliefbase.my_location.isNear($goal.location)
  </targetcondition>
</achievegoal>
```

Query Goals






```
<!-- Try to find a not full waste bin that
     is as near as possible to the agent. -->
<querygoal name="querywastebin" exclude="never">
  <parameter name="result" class="Wastebin" direction="out">
    <value evaluationmode="dynamic">
      select one Wastebin $wastebin
      from $beliefbase.wastebins
      where !$wastebin.isFull()
      order by
        $beliefbase.my_location.getDistance($wastebin.getLocation())
    </value>
  </parameter>
</querygoal>
```

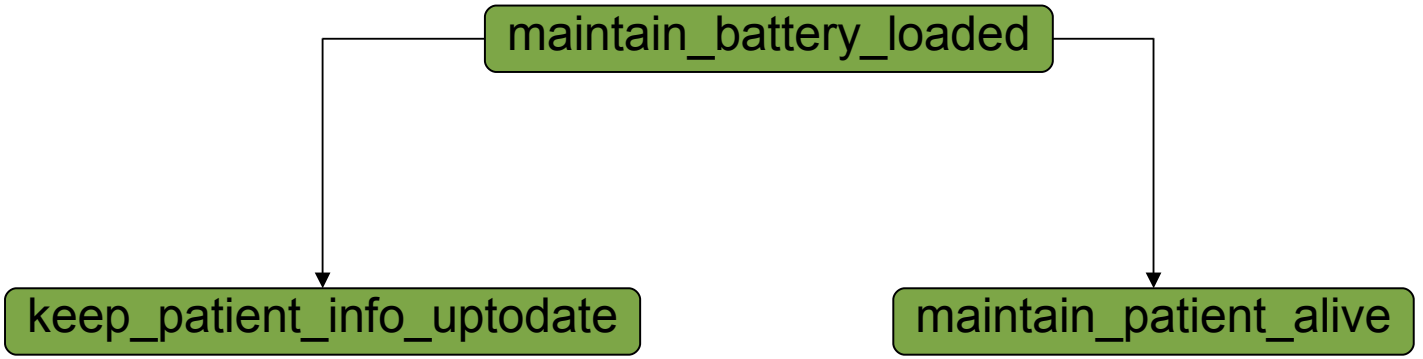
Conflicting Goals

- Goal-oriented agent is capable of pursuing multiple goals simultaneously
- Some goals could be in conflict
  - *Doctor* cannot take care about patient and regenerate its energy at the same time
- Some goals require limitation in number of activated instances
  - see *Cleaner* example in Jadex package

Goal Deliberation Strategy

- Goal deliberation allows avoiding activatation of conflicting goals
- Jadex uses Easy Deliberation strategy
  - *Cardinalities for goal instances*:
    Only x instances of a certain type of goal are allowed to be active simultanously
  - *Inhibition links*:
    Goals which has been activated should suspend goals inhibited by them

Inhibition Links



- **Idle** maintain goals (mainly them), might not always be in conflict with other goals → is sometimes required to restrict the inhibition to only take effect when the goal is in process.
- This can be specified with the **`inhibit`** attribute of the **`<inhibits>`** tag, using **`"when_active"`** (default) or **`"when_in_process"`** as appropriate.

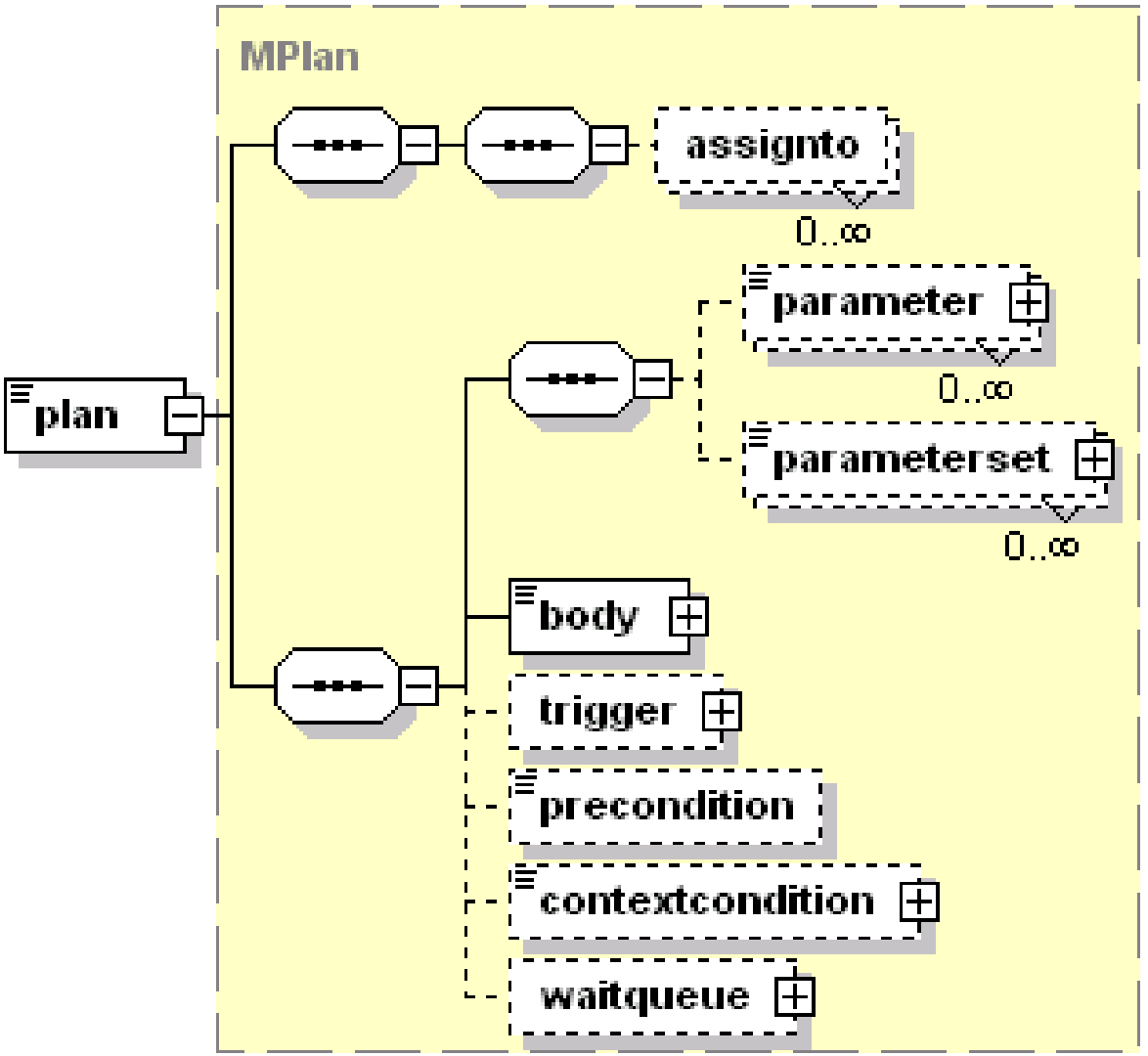
Inhibition Links

```
!-- Observe the battery state. -->
<maintaingoal name="maintain_battery_loaded" exclude="never"
retry="true">
    <deliberation>
    <!-- The Doctor's first takes care about its energy, does it
        cannot do anything else when regenerating. -->
        <inhibits ref="keep_patient_info_uptodate"
            inhibit="when_in_process"/>
        <inhibits ref="maintain_patient_alive"
            inhibit="when_in_process"/>
    </deliberation>

    <!-- Engage in actions when the state is below 20. -->
    <maintaincondition> $beliefbase.my_chargestate >= 20
    </maintaincondition>
    <!-- The goal is satisfied when the charge state is 100. -->
    <targetcondition> $beliefbase.my_chargestate >= 100
    </targetcondition>
</maintaingoal>
```

Goal Deliberation Strategy

- Graph consitisting of inhibiting arcs should be acyclic to avoid cycles in deliberations.
- Agent should deliberate only on demand:
  - *Deliberate a new option*
    Check which inhibited goals should be suspended.
  - *Deliberate a deactived goal*
    Check which inhibited goals should be reactivated.

Plan Head


MPlan
plan
assignto
0..∞
parameter
0..∞
parameterset
0..∞
body
trigger
precondition
contextcondition
waitqueue

Plan Head

- Create plan instance when a message arrives (plan pre-condition)

```
<!-- A  plan, from which a new instance is created
        whenever a drug applying request is received. -->
<plan name="apply_doctors_order">
    <body>new ApplyDoctorsOrderPlan()</body>
    <trigger>
        <messageevent ref="request_drug_applying"/>
    </trigger>
</plan>
```

Plan Head

- Create plan instance when a goal is adopted

```
<!-- Load the battery. -->
<plan name="load_battery">
    <body>new LoadBatteryPlan()</body>
    <trigger>
        <goal ref="maintain_battery_loaded"/>
    </trigger>
</plan>
```

Plan Head

- **`<precondition>`** is evaluated before a plan is instantiated
- When it is not fulfilled this plan is excluded from the list of applicable plans.

Plan Head

- **<contextcondition>** is evaluated before & during the execution of plans.
- When context condition is violated, the plan is aborted and the plan had failed.

```
<!-- Maintain correct patient's blood pressure, but only if Doctor
  has energy. -->
<maintaingoal name="maintain_patient_alive" exclude="never"
  retry="true" retrydelay="2500">
    <contextcondition>$beliefbase.my_chargestate >
        0</contextcondition>
<!-- Engage in actions when the pressure is out of [50,100] range. -->
    <maintaincondition> $beliefbase.patient_pressure >= 50 &amp;&amp;
        $beliefbase.patient_pressure &lt;= 100
    </maintaincondition>
</maintaingoal>
```

Plan Body

- The standard plans inherit from **`jadex.runtime.Plan`**
- This class provides set of abstract methods:
  - **`body()`** – plan code
  - **`passed()`** – optional cleanup code in case of a plan success
  - **`failed()`** – optional cleanup code in case of a plan failure
  - **`aborted()`** – optional cleanup code in case the plan is aborted
- Plan body may:
  - Send / receive messages
  - Manipulate beliefs
  - Create subgoals

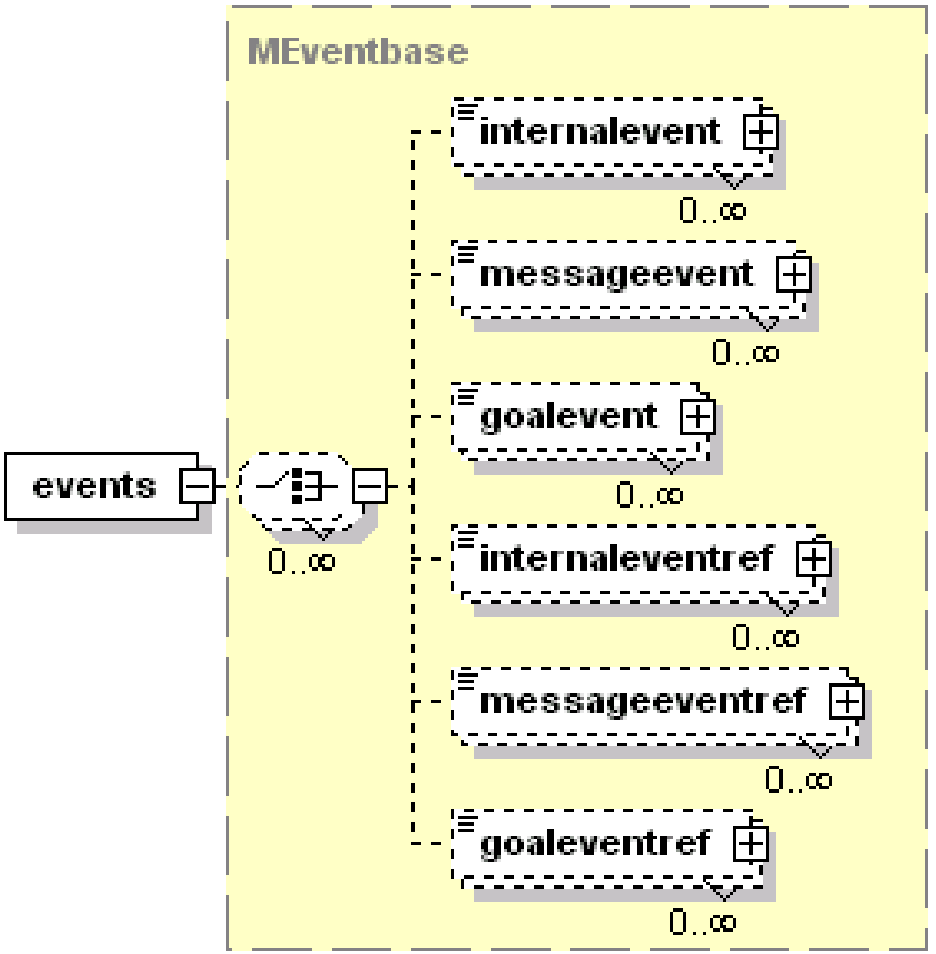
Plan Execution

- For the first step:
  - The **`body()`** method of standard plans is called only once
  - and runs until
    - the plan explicitly ends its step by calling one of the **`waitFor()`** methods,
    - or the execution of the plan triggers a condition (e.g., by changing belief values).
- For subsequent steps the **`body()`** method is continued, where the plan was interrupted.

Plan Execution

```
AgentIdentifier nurse = ...
if (nurse == null) {
    // If Nurse unknown yet, find it in Directory Facilitator
    IGoal df_search = createGoal("df_search");
    ...
    dispatchSubgoalAndWait(df_search);
    AgentDescription[] result = (AgentDescription[]) df_search
        .getParameterSet("result").getValues();
    ...
}

IMessageEvent outcoming = createMessageEvent("query_for_patient");
IMessageEvent incoming = sendMessageAndWait(outcoming);
Integer pressure = (Integer) incoming.getContent();
...
```

Events


MEventbase
internalevent
0..∞
messageevent
0..∞
goalevent
0..∞
internaleventref
0..∞
messageeventref
0..∞
goaleventref
0..∞
events
0..∞

Receiving Messages

- Incoming messages are handled by the event dispatching mechanism
- *Event dispatching mechanism* is based on two mappings:
  - from message to message event
  - from (message event) to plan trigger
- *Mappings* are recommended to be unambigous
- When more than one mapping from a received message to different *message events* are available
  - agent chooses the alternative which is the most specific
  - if there are two or more with the same specificity, the first one is chosen

Receiving Messages

- The message event (**`jadex.runtime.IMessageEvent`**) denotes the arrival or sending of a message.

Receiving Messages

```
<events>
    ...
    <!-- Specifies a drug applying request being all
    messages with performative request. -->
    <messageevent name="request_drug_applying" direction="receive"
            type="fipa">
        <parameter name="performative" class="String"
                direction="fixed">
            <value>SFipa.REQUEST</value>
        </parameter>
        <parameter name="language" class="String" direction="fixed">
            <value>SFipa.JAVA_XML</value>
        </parameter>
    </messageevent>
</events>

<plans>
    <!-- A  plan, from which a new instance is created
        whenever a drug applying request is received. -->
        <plan name="apply_doctors_order">
            <body>new ApplyDoctorsOrderPlan()</body>
            <trigger>
                <messageevent ref="request_drug_applying"/>
            </trigger>
        </plan>
    ...
</plans>
```

Receiving Messages

```
public class ApplyDoctorsOrderPlan extends Plan {
    ...
    public void body() {
        // Access the event that triggered this plan.
        IMessageEvent incoming = (IMessageEvent) getInitialEvent();
        // Get Doctor's order/decision.
        String decision = (String) incoming.getContent();
        ...
    }
```

Sending Messages

```
<events>
    ...
    <!-- Specifies a drug applying request being all
    messages with performative request. -->
    <messageevent name="request_drug_applying" direction="send"
            type="fipa">
        <parameter name="performative" class="String">
            <value>SFipa.REQUEST</value>
        </parameter>
        <parameter name="conversation-id" class="String">
            <value>SFipa.createUniqueId($scope.getAgentName())</value>
        </parameter>
        <parameter name="language" class="String">
            <value>SFipa.JAVA_XML</value>
        </parameter>
    </messageevent>
</events>
```

Sending Messages

```
public class DiagnosePatientPlan extends Plan {
    ...
    public void body() {
        ...
        // Prepare a message to the Nurse
        IMessageEvent outcoming =
            createMessageEvent("request_drug_applying");
        outcoming.getParameterSet(jadex.adapter.fipa.SFipa.RECEIVERS)
            .addValue(nurse);

        // Prepare diagnosis and decision to apply in the situation
        Integer pressure = (Integer)
            getBeliefbase().getBelief("patient_pressure").getFact();
        String decision = (pressure < 50) ?
            "DO_INJECTION" : (pressure > 100) ? "DO_VALIUM" : null ;
        if (decision != null) {
            outcoming.setContent(decision);
            IMessageEvent incoming = sendMessageAndWait(outcoming);
            ...
        }
    }
}
```

Replying Messages

```
<events>
    ...
    <!-- Specifies a return message about patient's blood pressure
        or being alive information, being all messages with
        performative inform. -->
    <messageevent name="inform_about_patient" direction="send"
            type="fipa">
        <parameter name="performative" class="String"
                direction="fixed">
            <value>SFipa.INFORM</value>
        </parameter>
        <parameter name="language" class="String" direction="fixed">
            <value>SFipa.JAVA_XML</value>
        </parameter>
    </messageevent>
</events>
```

```
public class InformAboutPatientPressurePlan extends Plan {
    ...
    public void body() {
        // Access the event that triggered this plan.
        IMessageEvent incoming = (IMessageEvent) getInitialEvent();
        Integer pressure = (Integer)
            getBeliefbase().getBelief("pressure").getFact();
        // Prepare reply of "inform_about_patient" type, defined in ADF
        IMessageEvent outcoming =
            incoming.createReply("inform_about_patient",pressure);
        sendMessage(outcoming);
```

Overview

- Theoretical foundation of BDI
- Introduction to Jadex reasoning engine
- Developing tools in Jadex
- JADE example
- Implementation in Jadex
- Conslusions

Documentation

- Jadex support
  - *Tutorial* and *User Guide*: http://vsis-www.informatik.uni-hamburg.de/projects/jadex/
  - forum and mailing list: http://sourceforge.net/projects/jadex
- Other presentations about Jadex
  - Prof. Michael N. Huhns, *Jadex and BDI Agents*: http://www.cse.sc.edu/~huhns/csce590/BDI-agents.ppt
  - Mehdi Dastani, *Multi-Agent Programming, Jadex: A BDI Reasoning Engine*: http://www.cs.uu.nl/docs/vakken/map/slides/jadex.pdf

Jadex summary

- Objective: Supporting the construction of open multiagent systems by making use of mentalistic notions
- Supports easy agent construction with XML-based agent description and procedural plans in Java
- Supports reusability through the capability concept offers toolsupport for debugging (in addition to the JADE tools)
  - BDI-Viewer allows to observe and modify the internal state
  - The BDI-Introspector allows to control the agent
  - The Logger agent collects log-outputs of any agents

FAQ

- *In my agents there is always one* plan *for a* goal*. Why do I need goals anyway?*
  - You don't need to use goals for every problem.
  - Using goals in many cases simplifies the development and allows for easier extensions of an application.
  - The difference between plans and goals is fundamental.
    - Goals represent the "what" is desired
    - plans are characterized by the "how" could things be accomplished.
  - If you e.g. use a goal "achieve happy programmers" you did not specify how you want to pursue this goals. One option might be the increase of salary, another might be to buy new TFT monitors.

FAQ

- *In my agents there is always one* plan *for a* goal*. Why do I need goals anyway?*
  - Example from Nurses

FAQ

- *How can the environment of a Jadex MAS be programmed?*
  - As a a separate environment agent:
    - Works when distribution required
    - The agent administers the environment
    - Domain specific ontology is defined: FIPA-compliant actions (e.g.such as *moveup*)
    - Each agent encodes each action into an *AgentAction*.
    - The environment agent tries to execute the contained action and sends back the result e.g. *Done(AgentAction)*.
    - As this procedure is cumbersome, we used following idea. For every primitive action a goal is defined with corresponding plans that do the message handling. The agent programmer can subsequently use just the goals for interaction with the environment.

FAQ

- *How can the environment of a Jadex MAS be programmed?*
  - As a singleton object for all agents:
    - Precisely as a simple belief with a fact expression that refers to that singleton object:
    - e.g. *garbagegollector* example

```
<!-- Environment object as singleton.-->
<belief name="env" class="Environment">
    <fact>Environment.getInstance($agent.getType(),
            agent.getName())</fact>
</belief>
```

    - Limited in nature as it is not possible to distribute the application over more than one Java VM.

Tools

- *XMLBuddy* plugin for *Eclipse*
  http://www.xmlbuddy.com/
  for editing XML files

Many Thanks Go To...

- Mehdi Dastani and Michael N. Huhns for their presentations
- Marcin Paprzycki and Maria Ganzha for their valuable comments

Thank you
for your attention