BDI programming

Stéphane Airiau[1]

[1]LAMSADE – Université Paris Dauphine

Later slides on JACK by Sebastian Sardina
RMIT University, Melbourne

An intelligent (software) agent is an autonomous entity, existing over time in a dynamic environment, that is able to rationally balance pro-active and reactive behavior.

- **autonomous:** does not require continuous external control;
- **pro-active:** pursues goals over time; goal directed behavior;
- **situated:** observe & act in the environment;
- **reactive:** perceives the environment and responds to it.
- Other features: flexible, robust, social, etc.

The Intentional Stance

Two major ideas:

1. Rational behavior can be understood in terms of mental properties (i.e., propositional attitude):
   - beliefs (“he thinks that Peter is wise”);
   - desires & goals (“she wants that piece of cake”);
   - fear (“Alex is afraid of spiders”);
   - hopes (“she hopes that he is on time today”);
   - ...
2. Rational behavior relies on a special kind of “thinking”
   - practical reasoning

Dennett coined the term **intentional system** to describe entities “whose behavior can be predicted by the method of attributing belief, desires and rational acumen.”

What is Practical Reasoning?

- Practical reasoning is reasoning directed **towards actions** – the process of figuring out what to do.
- Principles of practical reasoning applied to agents largely derive from work of philosopher Michael Bratman (1990):

  > *Practical reasoning is a matter of weighting conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes.*

The Components of Practical Reasoning

Human practical reasoning consists of two activities:

- **Deliberation:** deciding what state of affairs we want to achieve.
  - considering preferences, choosing goals, etc.;
  - balancing alternatives (decision-theory);
  - the outputs of deliberation are intentions;
  - interface between deliberation and means-end reasoning.
- **Means-ends reasoning:** deciding how to achieve these states of affairs:
  - thinking about suitable actions, resources and how to "organize" activity;
  - building courses of action (planning);
  - the outputs of means-ends reasoning are plans.

Fact: agents are resource-bounded & world is dynamic!

Key: To appropriately combine deliberation & means-ends reasoning.

Deliberation

How does an agent deliberate?

- Begin by trying to understand what the options available to you are:
  - options available are **desires**.
- Choose between them, and commit to some:
  - chosen options are then **intentions**.

$$\begin{tikzcd}\text{Desires} \arrow[r, dashed] & \begin{matrix}\text{choose}\\ \&\\ \text{Commit}\end{matrix} \arrow[r, dashed] & \text{Intentions}\end{tikzcd}$$

Desires

- Desires describe the states of affairs that are considered for achievement, i.e., **basic preferences** of the agent.
- Desires are much **weaker** than intentions; not directly related to activity:

> *"My desire to play basketball this afternoon is merely a potential influence of my conduct this afternoon. It must vie with my other relevant desires [...] before it is settled what I will do. In contrast, once I intend to play basketball this afternoon, the matter is settled: I normally need not continue to weigh the pros and cons. When the afternoon arrives, I will normally just proceed to execute my intentions."*

– (Bratman 1990)

Intentions

- In ordinary speech: intentions refer to **actions** or to **states of mind**;
  - here we consider the latter!
  - E.g., I may adopt/have the intention to be an academic.
- Focus on **future-directed** intentions i.e. **pro-attitudes** leading to actions.
  - Intentions are about the (desired) future.
- We make **reasonable attempts to fulfill** intentions once we form them, but they may change if circumstances do.
  - Behavior arises to fulfill intentions.
  - Intentions affect action choice.

- Intentions drive means-end reasoning.
- Intentions constrain future deliberation
- Intentions persist.
- Intentions influence beliefs concerning future practical reasoning.
- Agents believe their intentions are possible.
- Agents do not believe they will not bring about their intentions.
- Under certain circumstances, agents believe they will bring about their intentions
- Agents need not intend all the expected side effects of their intentions.
  So, intentions are not closed under implication!
  This last problem is known as the side effect or package deal problem: *" I may believe that going to the dentist involves pain, and I may also intend to go to the dentist but this does not imply that I intend to suffer pain!"*

Plans

Human practical reasoning consists of two activities:

- **Deliberation:** deciding what to do. Forms intentions.
- **Means-ends reasoning:** deciding how to do it. Forms plans. Forms plans.

Intentions drive means-ends reasoning: If I adopt an intention, I will attempt to achieve it, this affects action choice.

Means-End Reasoning: Obtaining Plans & Actions

How does the agent obtain plans/actions to realize our intentions?

- Planning: design a course of action that will achieve the goal. Given:
  - (representation of) goal/intention to achieve;
  - (representation of) actions it can perform; and
  - (representation of) the environment;

  ... have it generate a plan to achieve the goal. This is automatic programming This is hard (PSPACE-complete)!
- High-level programming (e.g., Golog, ConGolog, IndiGolog). This is semi-automatic/hybrid programming.
- BDI-style programming (e.g., AgentSpeak, CAN, Jason, JACK, etc.) This is implicit programming.

Commitments

We may think that deliberation and planning are sufficient to achieve desired behavior, unfortunately things are more complex...
Questions:

- how long should an intention persist?
- what is the commitment on?
  Commitments to Ends and Means.
- when to reconsider a commitment?
  (costly but necessary).

An agent has commitment both to ends (intentions), and means (plans).

Agent theory

Formal specifications of agent properties – what kind of mental states they have and how they are related to each other and to action; should support reasoning about agents.

Two major seminal works:

- Cohen & Levesque: "Intentions = Choice + Commitment"
- Rao & Georgeff's BDI logics: non-classical logics with modal connectives for representing beliefs, desires, and intentions.

- **Belief:** knowledge about the world and its own internal state
- **Desires (or goals):** what the agent has decided to work towards achieving
- **Intentions:** how the agents has decided to tackle these goals.
- **No planning from first principles:** agents use a plan library (library of partially instantiated plans to be used to achieve the goals)

Practical reasoning agents: quickly reason and react to asynchronous events.

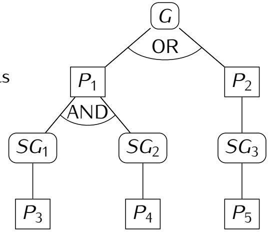
Definition (Plan)

A plan is $e : \psi \leftarrow P$ where

- e is an **event** that triggers the plan
- ψ is the **context** for which the plan can be applied
- P is the plan body (succession of actions and/or sub-goals)

Definition (Goal-Plan tree)

P_i: plan
G_i: goals
SG_i: sub-goals

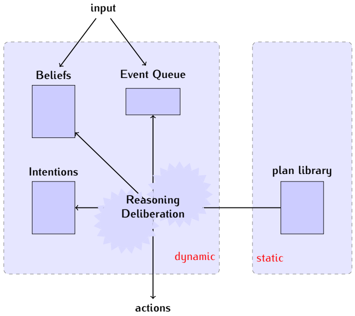


Definition (Failure recovery)

When a plan fails, an alternative plan is tried ex: $\mathcal{P}_1$ and $\mathcal{P}_2$ are both applicable. When $\mathcal{P}_4$ fails, $\mathcal{P}_2$ can be tried.

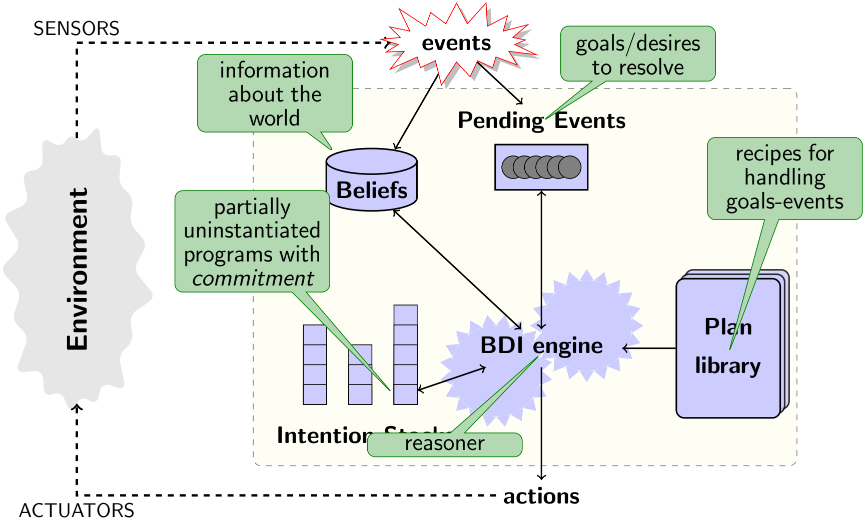
BDI execution algorithm

- Take the next event (internal/external)
- Update any goal, belief, intention (new event may cause an update of the belief ⇒cascading effect on goals or intentions)
- Select an applicable plan to respond to this event
- Place this plan in the intention base
- Take the next step on a selected intention (execute an action or generate a new event)



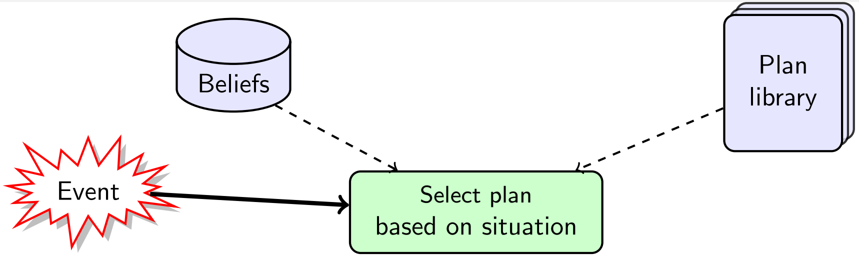
BDI agents are well suited for complex application with soft real-time reasoning and control requirements.

Detailed BDI Architecture

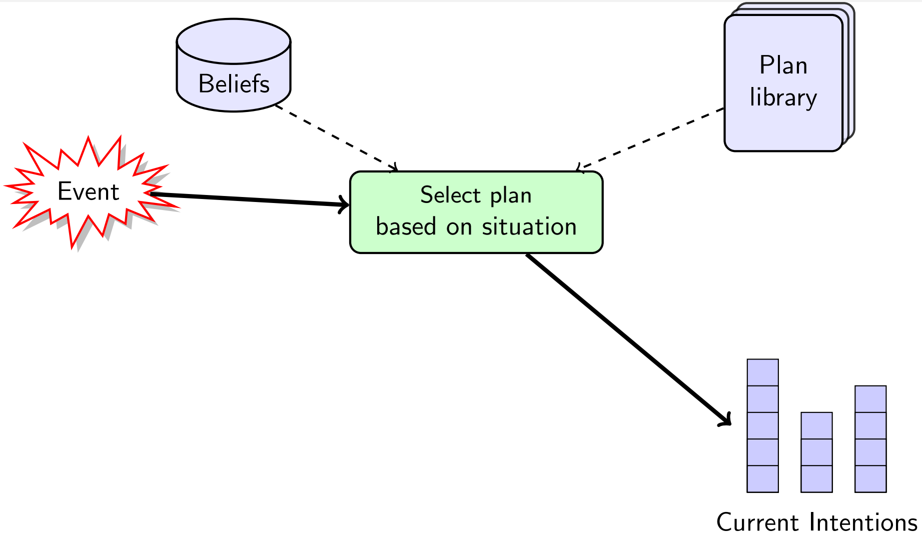


The BDI Execution Cycle [Rao&Georgeff 92]

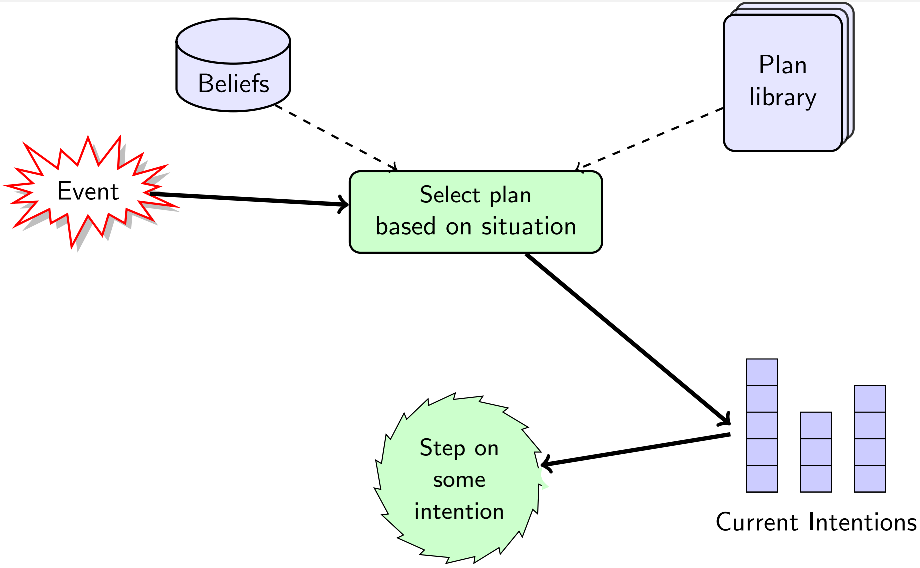
The BDI Execution Cycle [Rao&Georgeff 92]



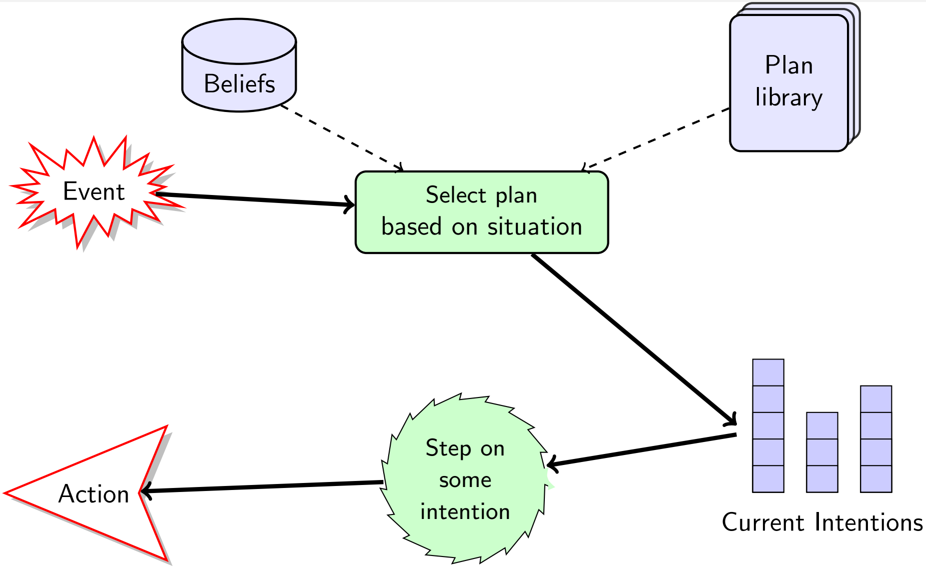
The BDI Execution Cycle [Rao&Georgeff 92]



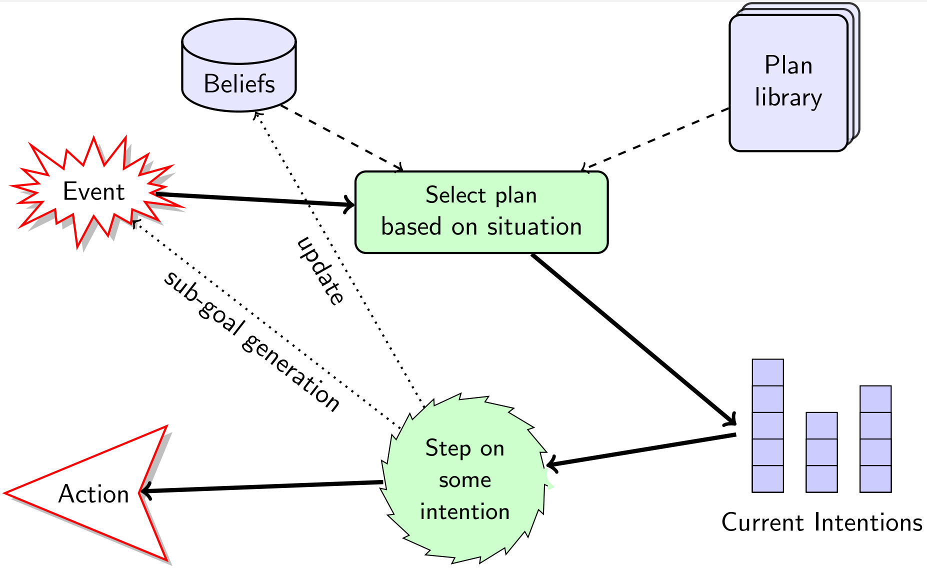
The BDI Execution Cycle [Rao&Georgeff 92]



The BDI Execution Cycle [Rao&Georgeff 92]



The BDI Execution Cycle [Rao&Georgeff 92]



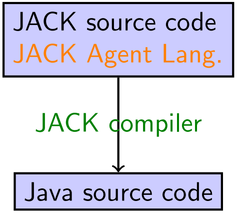
The JACK BDI Programming Language

1. JACK Agent Language
   - Used to describe an agent-oriented software system.
   - Super-set of Java (agent-oriented features extensions).

JACK source code
JACK Agent Lang.

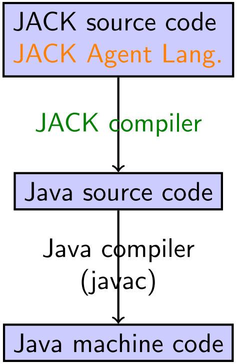
The JACK BDI Programming Language

1. JACK Agent Language
   - Used to describe an agent-oriented software system.
   - Super-set of Java (agent-oriented features extensions).
2. The JACK Agent Compiler
   - Converts JACK Agent Language into pure Java.
   - Java source can be compiled into Java VM code.



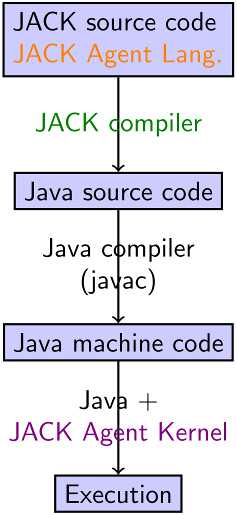
The JACK BDI Programming Language

1. JACK Agent Language
   - Used to describe an agent-oriented software system.
   - Super-set of Java (agent-oriented features extensions).
2. The JACK Agent Compiler
   - Converts JACK Agent Language into pure Java.
   - Java source can be compiled into Java VM code.

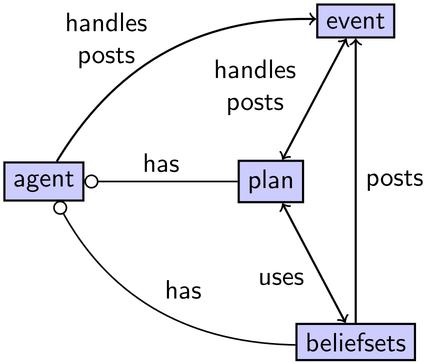


The JACK BDI Programming Language

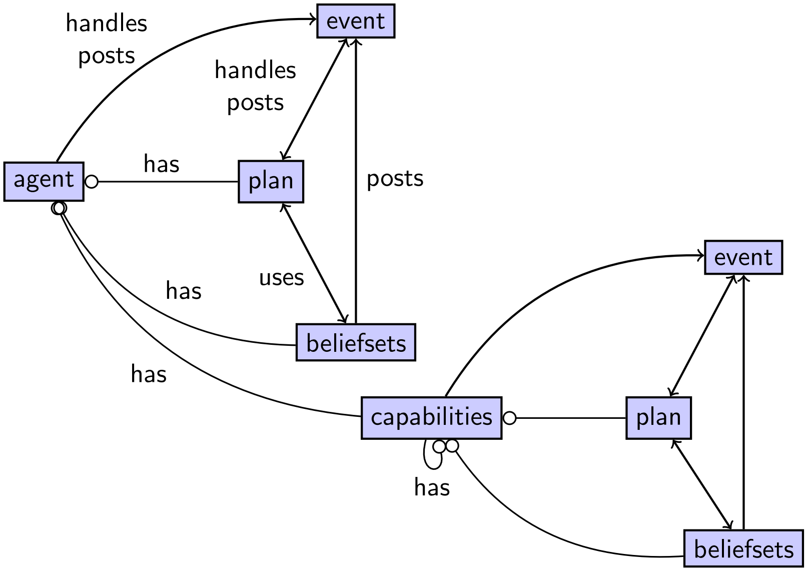
1. JACK Agent Language
   - Used to describe an agent-oriented software system.
   - Super-set of Java (agent-oriented features extensions).
2. The JACK Agent Compiler
   - Converts JACK Agent Language into pure Java.
   - Java source can be compiled into Java VM code.
3. The JACK Agent Kernel
   - Runtime engine for programs written in the JACK Agent Language.
   - Set of classes that give JACK Agent Language programs their agent-oriented functionality.
   - Run behind the scenes.
   - Implement the underlying infrastructure and functionality for agents.



Agent Components



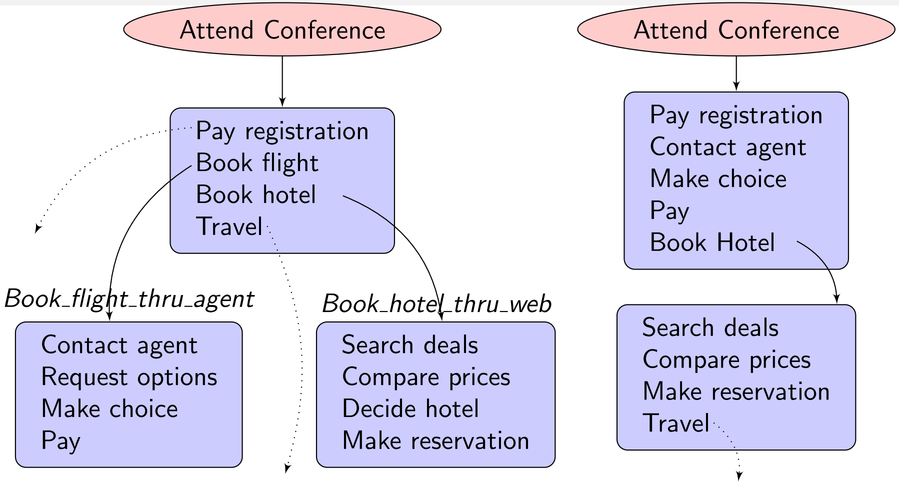
Agent Components



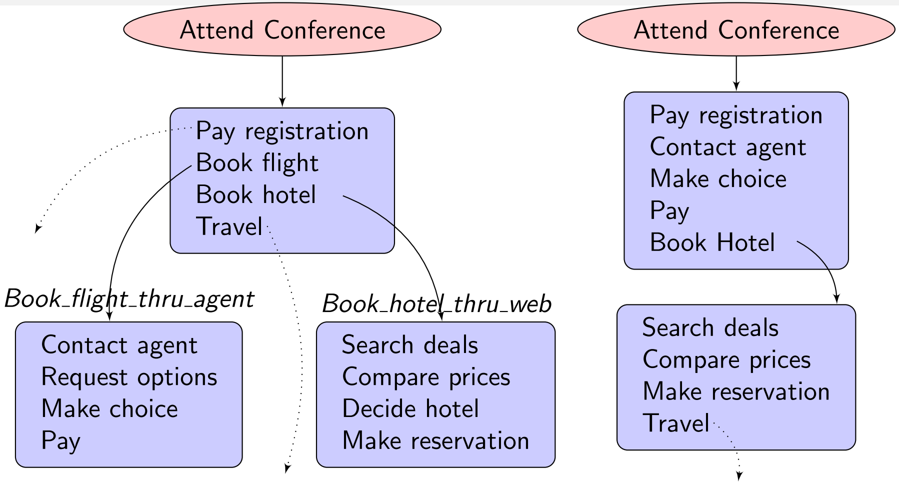
Making Use of the BDI Framework

1. Provide alternative plans where possible.
2. Break things down into subgoal steps.
3. Use subgoals and alternative plans rather than **if**... **then** in code.
4. Keep plans small and modular.
5. Plans are abstract modules - don't chain them together like a flowchart!

Plan Structure: Which one is better?



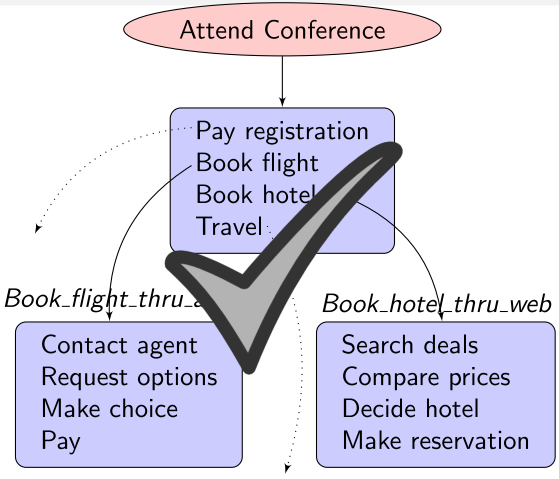
Plan Structure: Which one is better?

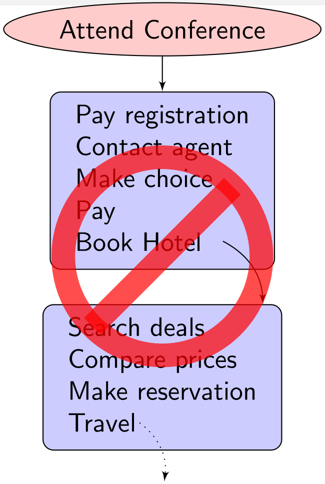


Hierarchical Structure
each plan complete at its level of abstraction

Chained Structure
do stuff and call next step

Plan Structure: Which one is better?





Hierarchical Structure
each plan complete at its level of abstraction

Chained Structure
do stuff and call next step

Structuring Plans and Goals

1. Make each plan complete at a particular abstraction level.
   - A high-level but complete plan for *Attend_Conference*.
2. Use a subgoal - even if only one plan choice for now.
   - Decouple a goal from its plans.
3. Modular and easy to add other plan choices later.
   - Booking a flight can now be done with the Internet, if available!
4. Think in terms of subgoals, not function calls.
   - What way-points do we need to achieve so as to realize a goal?
5. Learn to pass information between subgoals.
   - How are these way-points inter-related w.r.t. data?

Defining Agents in JACK: `Player.agent`

Base class: `aos.jack.jak.agent.Agent`

```
public agent Player extends Agent {
    #has capability ClimaTalking cap;
    #handles event PerceiveClimaServer;
    #handles event EExecuteCLIMAaction;
    #handles event EAct;
    #posts event EExecuteCLIMAaction ev_executeAction;
    #sends event EInformLoc ev_informLoc;
     ...
    #uses plan MoveRandomly;
    #uses plan PickGold;
    #uses plan HandlePercept;
     ...
    #private data GoldAt bel_goldAt();
    #private data CurrentPosition bel_currPosition();
    #private data NumCarryingGold bel_noCarrGold();
    ....
}
```

Beliefsets

- Used to maintain an agent's beliefs about the world.
- Represents these beliefs in a first order, tuple-based relational model.
- Designed specifically to work within the agent-oriented programming paradigm & integrated with the other JACK Agent Language classes:

  1. Automatic maintenance of logical consistency and key constraints.
  2. Either `OpenWorld` or `ClosedWorld` logic semantics.
  3. Ability to post events automatically when a beliefset changes.
     - initiate action within the agent based on a change of beliefs.
  4. Ability to support beliefset cursor statements: multiple query bindings.

Beliefset Definitions

```
beliefset RelationName extends <ClosedWorld | OpenWorld > {
        // Zero or more #key field declarations.
        // Zero or more #value field declarations
        // One or more queries
}
```

1. Type of beliefset:
   - `CloseWorld`: close-world-assumption; standard databases.
   - `OpenWorld`: allows for “unknown” values.
2. Fields of beliefset (i.e., columns in a database):
   - `#key field:` uniquely identifies tuples; `int`, `float`, `boolean`, or `String`.
   - `#value:` stores extra info. about the tuple; could be of any type.
3. Queries available:
   - `#indexed query`: $>$ 10 tuples
   - `#linear query`: for small beliefsets

Events in BDI Systems

- They encode the goals of the system at different times.
- Pending events are expected to be addressed eventually....
  - by executing some plan that "solves" them...
- Events can be either:
  - External: coming from outside the system (e.g., perception).
  - Internal: generated by the agent itself (e.g., subgoals).
- Events are often used to represent:
  - percepts: `PerceiveClimaTalking`
  - communication: `EGUIUpdateLocation`
  - internal goals: `EAct`

Events in JACK

- There are several type of events that can be defined:
  1. BDIFactEvent
  2. BDIGoalEvent
  3. BDIMessageEvent
  4. InferenceGoalEvent
  5. PlanChoice
  6. ...
- Different type of events depend on:
  - Allow meta-level reasoning: *can we tweak plan choices?*
  - Allow plan-failure recovery: *what happens when a plan fails for an event?*

Events in JACK

- There are several type of events that can be defined:
  1. BDIFactEvent
  2. BDIGoalEvent
  3. BDIMessageEvent
  4. InferenceGoalEvent
  5. PlanChoice
  6. ...
- Different type of events depend on:
  - Allow meta-level reasoning: *can we tweak plan choices?*
  - Allow plan-failure recovery: *what happens when a plan fails for an event?*
- Events can carry data inside:
  - `GoTo(destination)`
  - `PerceiveClimaTalking` events carries a whole object depending on the XML message sent by the game server!

Events in JACK

- There are several type of events that can be defined:
  1. BDIFactEvent
  2. BDIGoalEvent
  3. BDIMessageEvent
  4. InferenceGoalEvent
  5. PlanChoice
  6. ...
- Different type of events depend on:
  - Allow meta-level reasoning: *can we tweak plan choices?*
  - Allow plan-failure recovery: *what happens when a plan fails for an event?*
- Events can carry data inside:
  - `GoTo(destination)`
  - `PerceiveClimaTalking` events carries a whole object depending on the XML message sent by the game server!
- Events can be posted/generated by plans:
  - `@subtask(...)`: agent handles it synchronously as part of the same task.
  - `@post(...)`: agent handles it asynchronously by the current task execution thread.

Events in JACK (cont.)

```
public event EAct extends BDIGoalEvent {
    public double deadline;  // how quickly we need to act

    #posted as postWithDeadline(double x) {
        deadline = x;
    }
    #posted as anytime() {
        deadline = 0;    // no deadline
    }
}
```

Events in JACK (cont.)

```
public event EAct extends BDIGoalEvent {
    public double deadline;  // how quickly we need to act

    #posted as postWithDeadline(double x) {
        deadline = x;
    }
    #posted as anytime() {
        deadline = 0;    // no deadline
    }
}
```

```
public plan XXXXX extends Plan {
    ...
    #posts event EAct ev_doSomething;
    ...
    #reasoning method body()    {
        ...
        @subtask(ev_doSomething.postWithDeadline(200));
        @post(ev_doSomething.anytime());
    }
}
```

Plans

Plans can be thought of as pages from a operational manual, or even as being like methods and functions from more conventional programming languages.

They describe "exactly" what an agent should do when a given event occurs.

Agent is equipped with a set of plans, describing the agent's set of skills.

When the event that a plan addresses occurs, the agent can execute this plan to handle it.

Plans

Plans can be thought of as pages from a operational manual, or even as being like methods and functions from more conventional programming languages.

They describe “exactly” what an agent should do when a given event occurs.

Agent is equipped with a set of plans, describing the agent’s set of skills.

When the event that a plan addresses occurs, the agent can execute this plan to handle it.

So, suppose the agent has the following BDI event:

```
// Prompts the player to act towards the game server
public event EAct extends BDIGoalEvent {

    #posted as   post() { }
}
```

Handling `EAct` BDI Event: Random Movement

```
public plan MoveRandomly extends Plan {
    final static String []
                actions = { "left", "right", "up", "down" };
    Random random = new Random();
    #handles event EAct ev_act;
    #posts event EExecuteAction ev_exec;

    static boolean relevant(EAct ev) {
        return true;
    }

    context()    {
        true;
    }

    #reasoning method body()    {
        @post(ev_exec.post(actions[ random.nextInt(4) ]));
    }
}
```

Handling `EAct` BDI Event: Random Movement

```
public plan MoveRandomly extends Plan {
    final static String []
              actions = { "left", "right", "up", "down" };
    Random random = new Random();
    #handles event EAct ev_act;
    #posts event EExecuteAction ev_exec;

    static boolean relevant(EAct ev)  {
        return true;
    }

    context()     {
        true;
    }

    #reasoning method body()     {
       @post(ev_exec.post(actions[ random.nextInt(4) ]));
    }
}
```

BDI Plan Selection

1. Identify the plans which handle the event type: `#handles event ..`
   - syntactic relevance.
2. Use the `relevant()` method to check additional information regarding the event.
   - inspect data carried on in the event.
3. Use the `context()` method to check information stored as part of the agent's beliefs.
   - defines the set of all applicable plans (types & instances).
4. All applicable plans are collected at this point.
5. If there are still multiple plans left in the applicable plan set, additional means are used to select one of them:
   - declaration order;
   - prominence w.r.t. plan ranks;
   - meta-level reasoning via `PlanChoice` handling.

Pick A Block I

```
public plan PickBlock extends Plan {

   #handles event EPickObject ev_pickObj;



   static boolean relevant(EPickObject ev)    {
      true;
   }

   context()    {
      true;
   }

   #reasoning method body()    {
      grabObj(ev_pickObj.x,ev_pickObj.y);
   }
}
```

Pick A Block II

```
public plan PickBlock extends Plan {

   #handles event EPickObject ev_pick;
   #posts event EExecuteAction ev_exec;


    static boolean relevant(EPickObject ev)    {
        return (ev.distance < 30);
    }

    context()    {
       true;
    }

   #reasoning method body()    {
       @post( ev_exec.post(grab,ev_pick.x,ev_pick.y) );
    }
}
```

Pick A Block III

```
public plan PickBlock extends Plan {

    #handles event EPickObject ev_pick;
    #posts event EExecuteAction ev_exec;
    #uses data Holding bel_holding;

    static boolean relevant(EPickObject ev)    {
        return (ev.weight < 30);
    }
    logical int $noObj;
    context()    {
        (bel_holding.get($noObj) && $noObj.as_int()<3);
    }

    #reasoning method body()    {
        @subtask( ev_exec.post(grab,ev_pick.x,ev_pick.y) );
    }
}
```

The Reasoning Method body()

Special kind of method in the JACK Language called a reasoning method.

Reasoning methods are quite different from ordinary methods in Java.

Each statement in a reasoning method is treated as a logical statement.

- Failure of a plan statement will cause the body() method to fail.
- If execution proceeds to the end, the body() method succeeds.

```
    #posts event EExecuteAction ev_execute;
    ...
    #reasoning method body()    {
        ...
        @subtask(ev_execute.do(actions[ "up" ]));
        ...
        @post(ev_execute.do(actions[ "pick" ]));
    }
```

The Reasoning Method body()

Special kind of method in the JACK Language called a reasoning method.

Reasoning methods are quite different from ordinary methods in Java.

Each statement in a reasoning method is treated as a logical statement.

- Failure of a plan statement will cause the body() method to fail.
- If execution proceeds to the end, the body() method succeeds.

```
#posts event EExecuteAction ev_execute;
...
#reasoning method body()    {
    ...
    @subtask(ev_execute.do(actions[ "up" ]));
    ...
    @post(ev_execute.do(actions[ "pick" ]));
}
```

The body() method can call *other* reasoning methods as it executes.
∴ Describe logical behavior that the reasoning method should adhere to.

Reasoning methods execute as Finite State Machines (FSMs).

Some Reasoning Methods

- `@subtask(event);`
- `@post(event);`
- `@send(agent_name, message_event);`
- `@wait_for(wait_condition);`
- `@action(parameters) <body>;`
- `@maintain(logical_condition, event);`
- `@reply(original_event, reply_event);`
- `@sleep(timeout);`
- `@achieve(condition, goal_event);`
- `@insist(condition, goal_event);`
- `@test(test_condition, goal_event);`
- `@determine(binding_condition, goal_event);`
- `@parallel(parameters) <body>;`

Example of enhancement of BDI tool

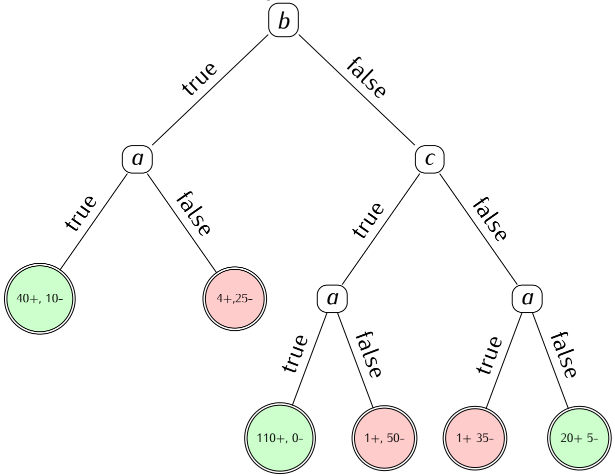
- BDI agents lack learning capabilities to modify their behavior (e.g. in case of frequent failures)
- Plans and context conditions are programmed offline by a user. Context conditions may be hard to capture precisely in a complex environment
  - too loose: plan is applicable when it is not ➯failures
  - too tight: plan is not applicable when it actually is ➯a goal may not appear achievable when it is

Research goal: Add learning capabilities to adapt and to refine context conditions of plans to the particular environment where the agent acts.

A first step: Use a decision tree (DT) in addition to the context condition
➯Each plan has a decision tree telling whether it is applicable

Example of a decision tree

The environment is described by three Boolean attributes *a*, *b* and *c*.



Context condition converted from the decision tree :
$(a \wedge b) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge \neg b \wedge \neg c)$.

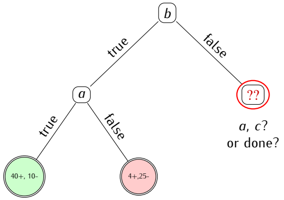
Induction of a decision tree

- Use a DT when
  - Instances describable by attribute-value pairs
  - Target function is discrete valued
  - Disjunction hypothesis may be required
  - Possibly noisy data
- **Inductive bias:** preference of a shorter tree
- Induction is performed **offline**
  Basic algorithm (AD3): top down, greedy algorithm (no search of an optimal tree)

Algorithm 1: ID3 main loop

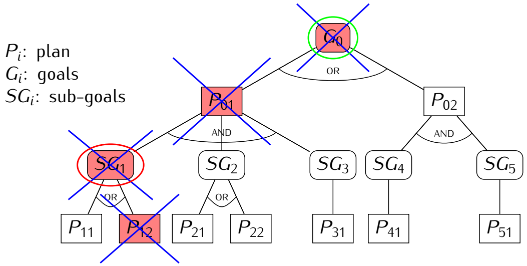
```
while exists a node n not marked do
    mark n;
    choose the best attribute att;
    for each value of att do
        create a new child of n;
        distribute the corresponding instance to
        each child of n;
    if training instances well classified then
        mark all children of n;
```

a	b	c	outcome
$\top$	$\top$	$\bot$	✔
$\top$	$\bot$	$\bot$	✘
$\top$	$\bot$	$\top$	✔
…	…	…	…
$\bot$	$\bot$	$\top$	✘



Issues with learning

- **Incremental induction:** algorithms exist but our current work uses offline algorithm and re-build a DT after a new instance is added to the dataset.
- **When to collect data?** A failure occurred during execution of $\mathcal{P}_{12}$ and propagates up (failure recovery OFF). Was the failure due to
  - a bad choice of plan for G_0 ($\mathcal{P}_{01}$ vs. $\mathcal{P}_{02}$)? ⇨Correct data
  - a bad choice after P01 was chosen ($\mathcal{P}_{11}$ vs. $\mathcal{P}_{12}$) ⇨Incorrect data



- **When to start using the decision tree?**

Assumptions

- Failure recovery is turned OFF.
- Do not consider effects of conflicting interactions between sub-goals.
- Domain described by Boolean attributes
- Domain may be stochastic (experiments involve actions that fail with a probability of 0.1)

Contributions

Techniques developed:

- Using a probabilistic plan selection for which plans are selected according to the frequency of success provided by the decision tree:
  trust the decision tree
  ➾carefully use of the data to induce a decision tree: add a failure in the dataset when we have some confidence that DT under are correct.
- Using a confidence degree about a decision tree for plan selection, and be less careful about the data to induce the decision tree.

How to test these strategies?
➾using three representatives goal-plan tree.

Algorithm 2: Probabilistic plan selection

```
Current world state is described by valued pair vector $\vec{s}$;
for each applicable plan $\mathcal{P}_i$ do
    retrieve the leaf node $n$ corresponding to $\vec{s}$;
    $n_\oplus \leftarrow$ number of positive outcomes contained in $n$;
    $n_\ominus \leftarrow$ number of negative outcomes contained in $n$;
    $p_i \leftarrow \frac{n_\oplus}{n_\oplus + n_\ominus}$;
Select plan $\mathcal{P}_i$ with a probability proportional to $p_i$;
```

Allows exploration.
A successful plan is **more likely** to be selected.

Algorithm 3: RecordTrace(λ, k, ϵ)

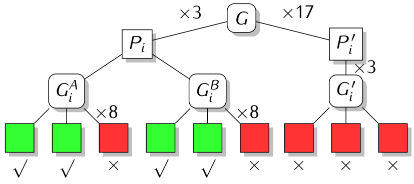
```
Requires: $\lambda = G_0[\mathcal{P}_0 : w_0] \cdot \cdots \cdot G_n[\mathcal{P}_n : w_n]$; $k \geqslant 0$; $\epsilon > 0$;
Ensures: Propagation of updates in the goal-plan tree.;

$RecordWorldDT(\mathcal{P}_n, w_n, fail)$ ;          // leaf node of gp-tree
if $StableGoal(G_n, w_n, k, \epsilon)$ then // this decision was well-informed
    // select relevant part of the trace
    $\lambda' \leftarrow G_0[\mathcal{P}_0 : w_0] \cdot \cdots \cdot G_n[\mathcal{P}_{n-1} : w_{n-1}]$;
    // call to the parent in the GP-tree
    $RecordTrace(\lambda', k, \epsilon)$
```

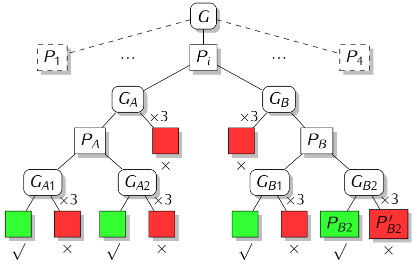
$StableGoal(G, w, k, \epsilon)$ is *true* when the frequency of success of goal G in state w has not changed by more than ϵ over the last k times G was used.

BUL (small and large k): cautious approach
(in the experiments $k = 3$, $\epsilon = 0.3$) Aggressive
Concurrent Learning (ACL) ($k = 0$, $\epsilon = 1$): always record, aggressive approach.
Initially designed to understand the benefits of BUL.

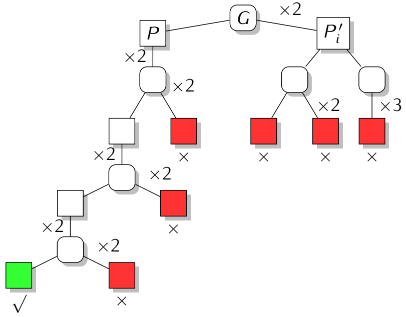
Goal-plan trees for testing



Structure $\mathcal{T}_1$: $ACL \succ BUL$

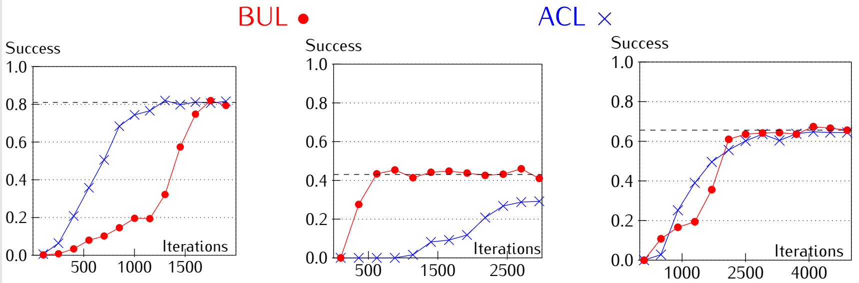


Structure $\mathcal{T}_3$: $ACL \approx BUL$



Structure $\mathcal{T}_2$: $BUL \succ ACL$

Results for Probabilistic Plan Selection



Structure $\mathcal{T}_1$ Structure $\mathcal{T}_2$ Structure $\mathcal{T}_3$

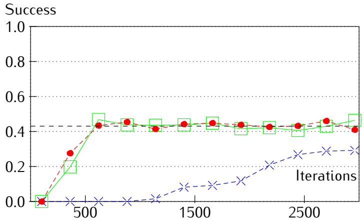
Plan selection with a degree of confidence of the DT

- **coverage $c_i(w)$ of a plan $\mathcal{P}_i$ in state w:** proportion of paths below $\mathcal{P}_i$ already visited from world state w .
  ➩initially $c_i(w) = 0$
  ➩$c_i(w) \to 1$: when more and more paths are tried
  ➩$c_i(w) = 1$: all paths have been tried.
  In a deterministic environment, behavior for w is known.
- **Weight for plan $\mathcal{P}_i$:** $\omega_i(w) = \frac{1}{2} + \left(c_i(w) \cdot \left(p_i(w) - \frac{1}{2}\right)\right)$
  ➩initially $\omega_i(w) = \frac{1}{2}$
  ➩weak recommendation $p_i(w) \approx \frac{1}{2}$ ➩$\omega_i(w) \approx \frac{1}{2}$
  ➩low coverage/confidence $c_i(w)$ small ➩$\omega_i(w) \approx \frac{1}{2}$ with slight bias towards recommendation
  ➩high coverage/confidence $c_i(w)$ large ➩$\omega_i(w) \approx p_i(w)$
- Select plan $\mathcal{P}_i$ with probability proportional to ω_i .
- Aggressive approach for recording data: ACL approach

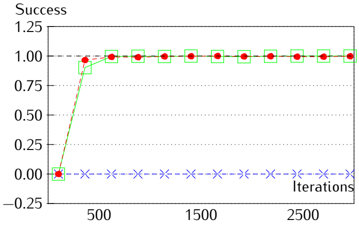
Results ACL with coverage vs. BUL with prob. plan selection

BUL ● ACL × ACL with coverage □

Structure $\mathcal{T}_2$



$\epsilon = 0.3$



$\epsilon = 0.2$

ACL with coverage performs as well as BUL with probabilistic selection.

Conclusion

- We proposed two techniques for tailoring context conditions to the environment where the agent resides
- Both BUL with probabilistic selection and ACL with coverage-based selection perform well on all tested structures
- ACL with coverage based is simpler

Future Work

- Using learning with recovery failure mode.
- Experiments with more difficult Decision trees to learn