Robotics and Autonomous Systems

Lecture 21: The Jason Interpreter

Richard Williams

Department of Computer Science
University of Liverpool

UNIVERSITY OF LIVERPOOL

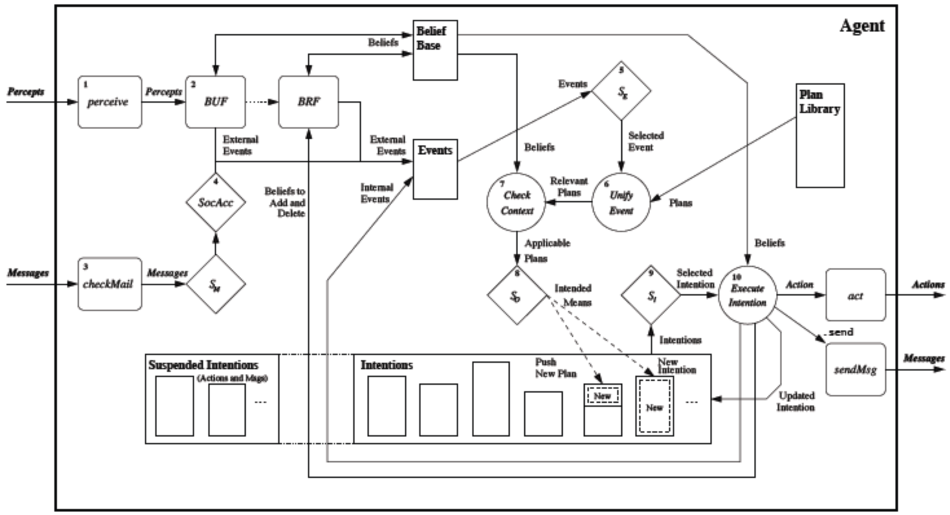
Today

- The previous couple of lectures have introduced the language and environment that you will use for the second assignment:
  - Jason
  - AgentSpeak
- This lecture will look at Jason in more detail.
- Understanding how Jason works will help you to know how to write AgentSpeak programs.

Programming in AgentSpeak

- Agent programs are written in AgentSpeak and consist of sets of goals, plans and beliefs
- How these components of the program interact to determine the actual behavior of the agent program is determined by
  - Jason: the interpreter
- The interpreter runs the agent implementing a reasoning cycle (= BDI decision loop)
- Understanding how the interpreter works, is key to programming agents in Jason

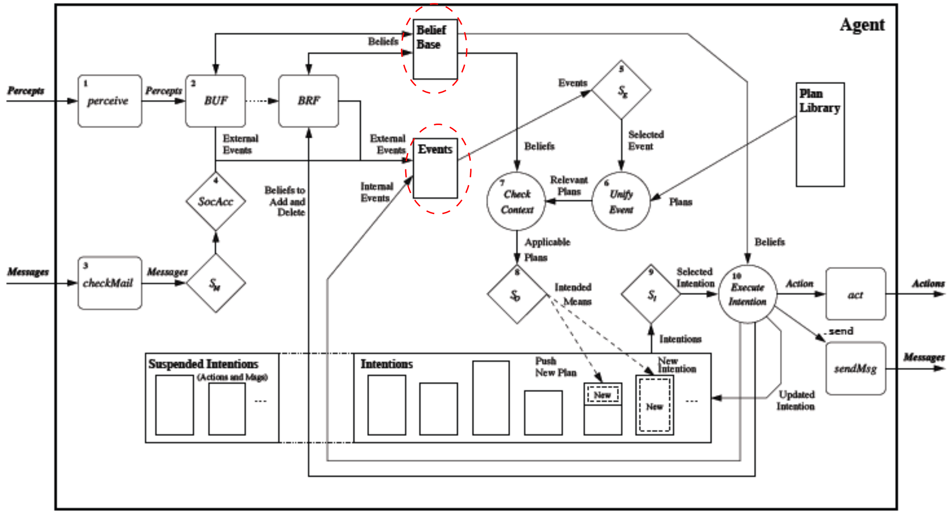
Main loop



Main loop

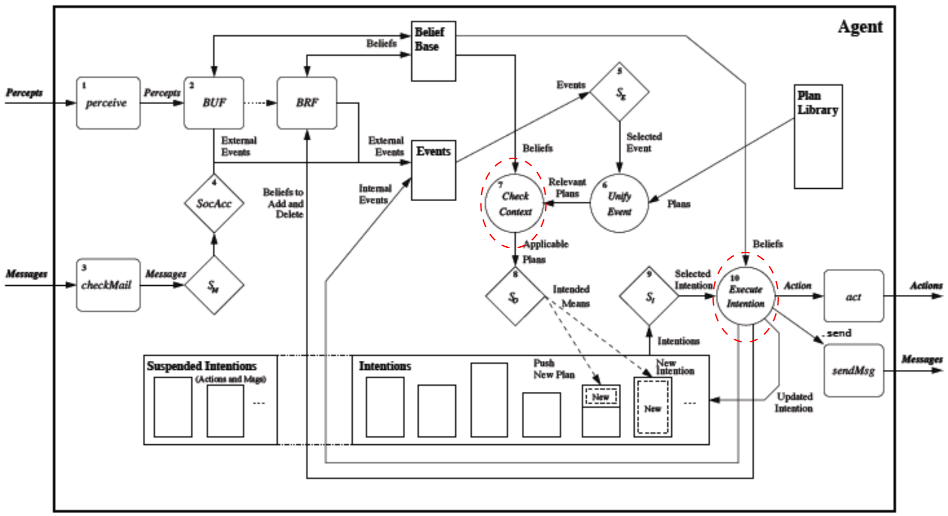
- Ok, so it is a bit more complex than the BDI cycle.
- Let's break it down a bit.

Main loop



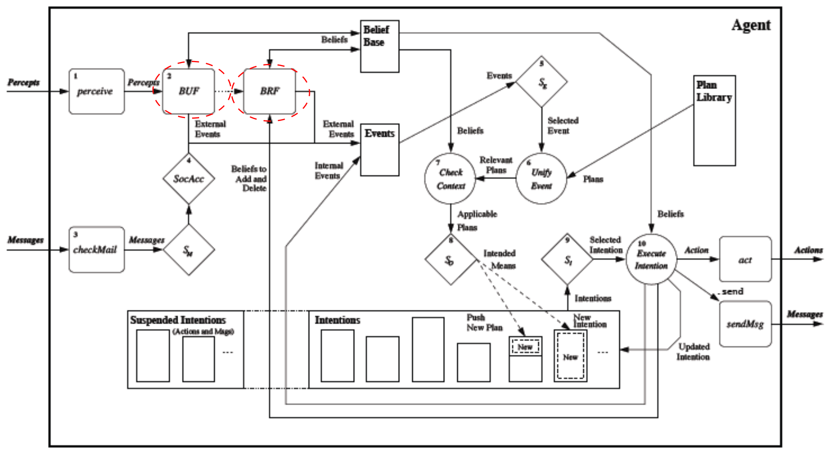
- Rectangles: basic components.

Main loop



- Circles: fixed methods of the interpreter

Main loop



- Rounded boxes: customizable methods

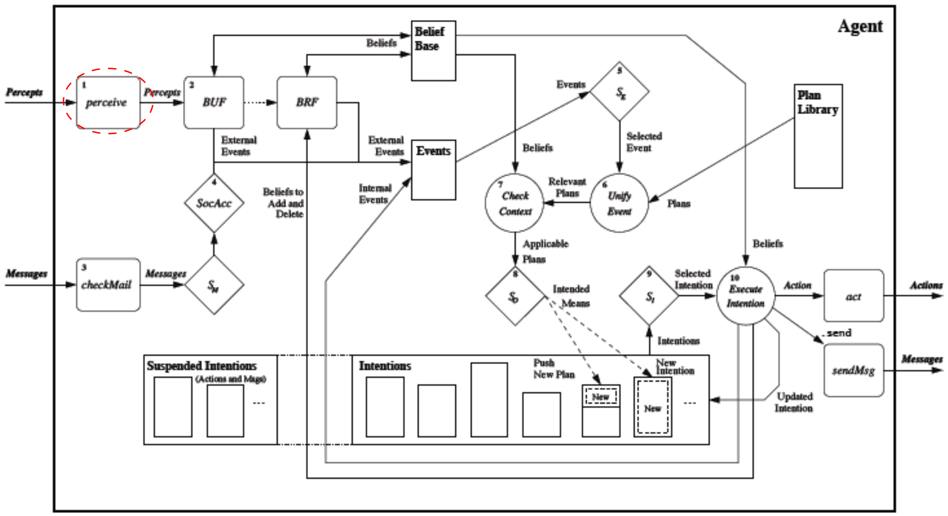
Main loop

- We will look at two bits of this in some detail:
  - Belief update
  - Event handling
- Event handling is basically everything you need to know about how programs are executed.

Belief update

- Captures how the agent changes its view of the world.
- Three components:
  - Perception
  - Belief revision
  - Incoming messages
- Will consider them in sequence.

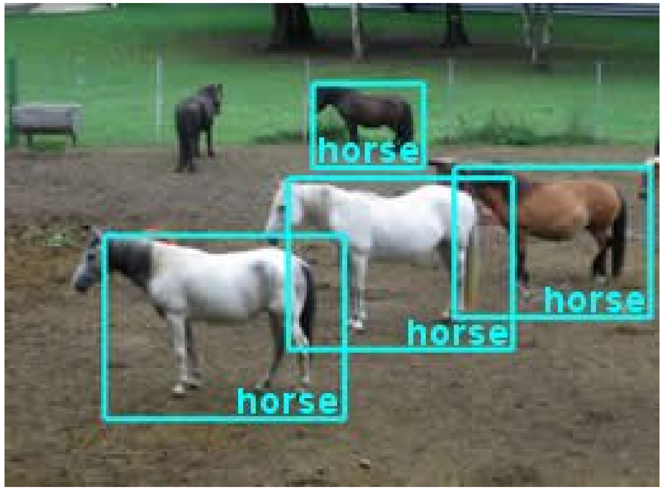
Perception



Perception

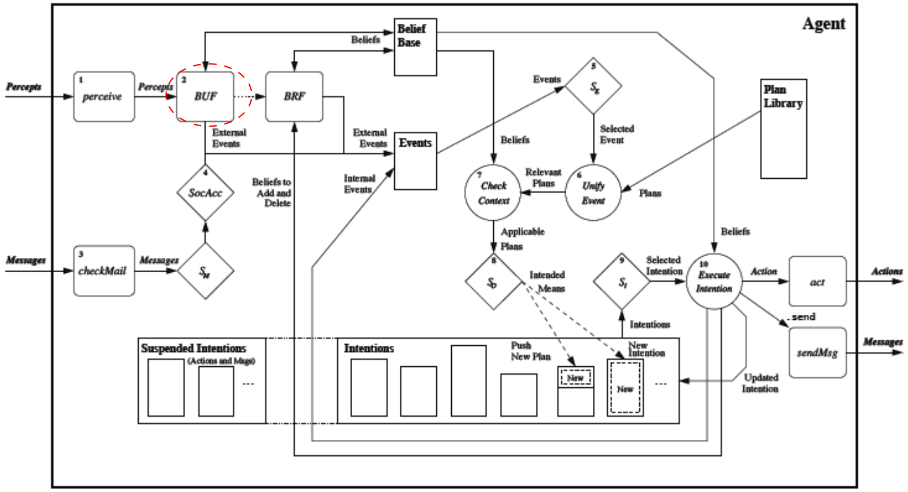
- Perception in Jason consists in the process of acquiring percepts consisting of logical literals.
- These are symbolic representation of the state-of-affairs being perceived
- They can be acquired via a simulated environment, or by interfacing real-world devices like robots
- The Perceive method implements this process by obtaining a list of literals (the percepts) from the environment
- To interface to the robot, you will have to supply this list of literals

Perception



- This involves translating sensor data into literals.

Belief Update Function



Belief Update Function

- Once the list of percepts has been obtained, the belief base needs to be updated
- BUF implements a default method for achieving that. Let **P** be the list of percepts and **B** the current belief base.
  - each literal in **P** but not in **B** is added to **B**
  - each literal in **B** no longer in **P** is removed from **B**
- Each such change generates an event (which may trigger a plan!).

Belief Update Function

- This approach to belief update involves enumerating all the beliefs.
- Not very efficient!

Belief Update Function

- Example update:
  $\langle +\mathtt{colour}(\mathtt{box1}, \mathtt{red})[\mathtt{source}(\mathtt{percept})], \top\rangle$
  - $+\mathtt{colour}(\mathtt{box1}, \mathtt{red})$ is the new belief
  - $[\mathtt{source}(\mathtt{percept})]$ says it came from perception
  - $\top$ says it is an external event.

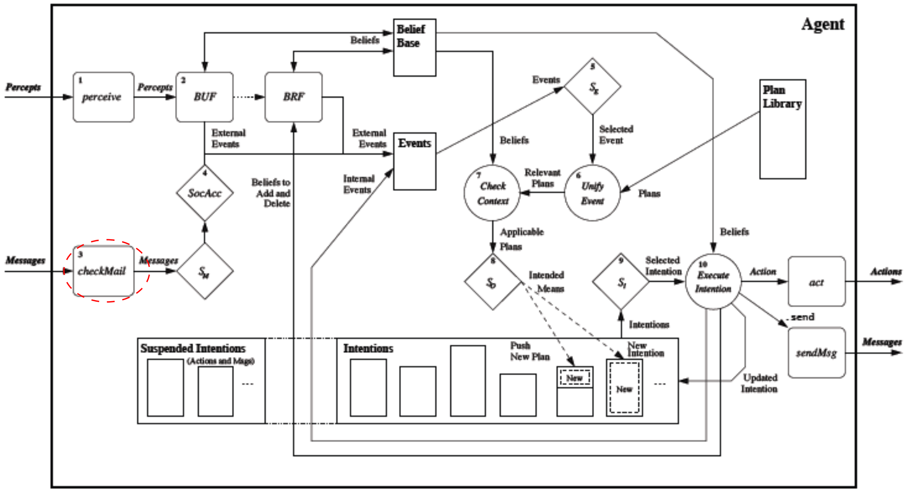
Belief Update Function

- If that box disappears:

$$\langle -\mathtt{colour}(\mathtt{box1}, \mathtt{red})[\mathtt{source}(\mathtt{percept})], \top\rangle$$

  is the update.

Messages from other agents

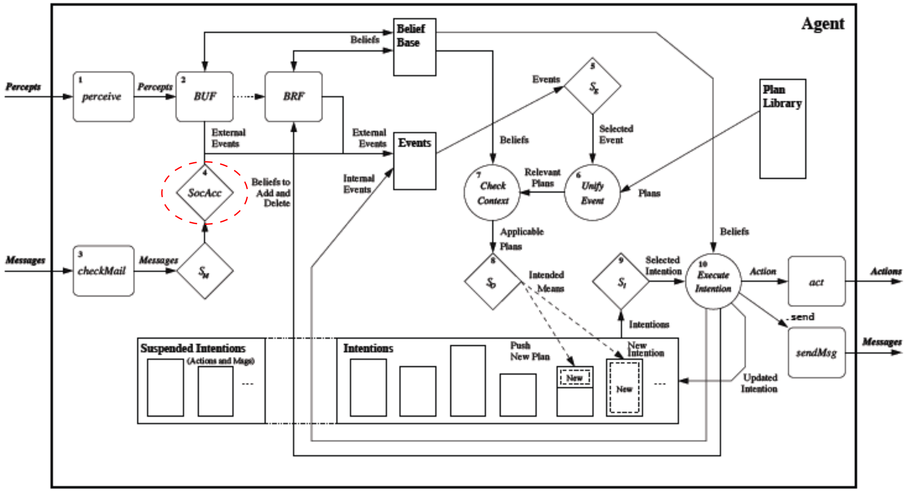


Messages from other agents

- Another source of information for agents are messages from other agents
- The checkMail method obtains messages for the agent (that are stored on the underlying multiagent system infrastructure)
- The messages may then be selected through a selection function (which is user-defined) in order to impose priorities upon them
- The default implementation just selects the first message in the queue
- Messages also generate events (annotate beliefs):

$$\langle +\mathtt{colour}(\mathtt{box1}, \mathtt{red})[\mathtt{source}(\mathtt{agent1})], \top\rangle$$

Socially acceptable messages

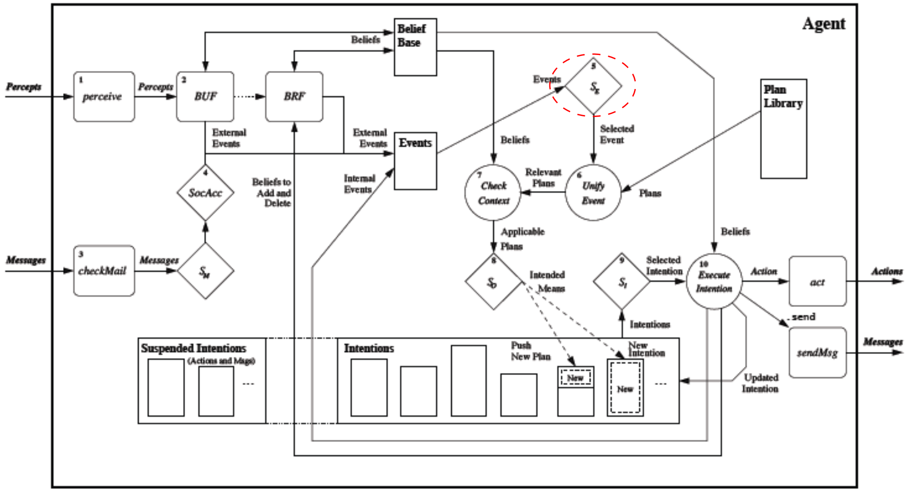


Socially acceptable messages

Socially acceptable messages

- The SocAcc method implements a social acceptance function which further filters incoming messages after their selection
  - filters according to criteria such as the “social structure” within a multiagent system
  - a sort of spam filter
  - allows an agent, for example, to ignore messages from a specific agent.
- This method is also typically customized by the user

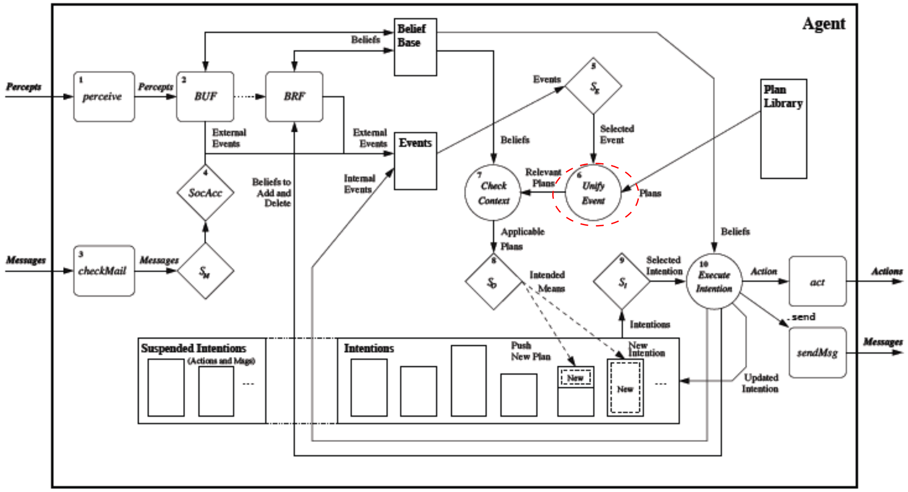
Event handling



Event handling

- BDI agents operate by reacting to events (they trigger plans!)
- In each reasoning cycle, only one pending event at the time can be handled
- This requires an event selection function operating on the set of pending events.
  - Intuitively, this selection function incorporates the “interests” of the agent, what they consider relevant
- The default implementation function handles events in a queue by a first-in first-out principle

Retrieving all relevant plans



Retrieving all relevant plans

- Once an event has been selected, relevant plans

  i.e., plans that can handle the event

  need to be retrieved
- This is done through a procedure called unification consisting of matching the “type” of the event.

Retrieving all relevant plans

- An example:

  $\langle +\texttt{colour}(\texttt{box1}, \texttt{blue})[\texttt{source}(\texttt{percept})], \texttt{T}$

  would match some of:

  $+\texttt{position}(\texttt{Object}, \texttt{Coords})$: ... < −....

  $+\texttt{colour}(\texttt{Object}, \texttt{Colour})$: ... < −....

  $+\texttt{colour}(\texttt{Object}, \texttt{Colour})$: ... < −....

  $+\texttt{colour}(\texttt{Object}, \texttt{red})$: ... < −....

  $+\texttt{colour}(\texttt{Object}, \texttt{Colour})[\texttt{source}(\texttt{self})]$: ... < −....

  $+\texttt{colour}(\texttt{Object}, \texttt{blue})[\texttt{source}(\texttt{percept})]$: ... < −....
- Which would it match?

Substitution and Unification

- A substitution is a function from a finite set of variables to a finite set of variables or constants. It can be viewed as a set of replacements:

  $\sigma = \{X_1 \rightarrow \chi_1, \dots, X_n \rightarrow \chi_n\}$

  where X_i are variables, and χ_i are variables or constants.
- Constraints:
  - $X_i \neq X_j, i \neq j$
  - $X_i \neq \chi_j, i \neq j$
- Example:

  $\sigma = \{X \rightarrow comp329\}$

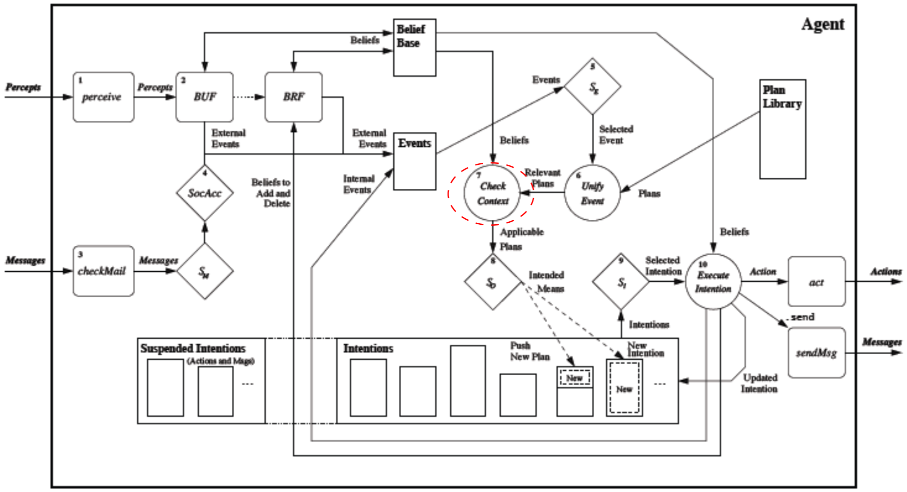
  $\sigma(\mathtt{lecturer}(\mathtt{X}, \mathtt{Y})) = \mathtt{lecturer}(\mathtt{comp329}, \mathtt{Y})$

Substitution and Unification

- A substitution for two formulae/predicates is a unifier iff the substitution applied to the two formulae/predicates yield the same result

  $\sigma(\mathtt{lecturer}(\mathtt{X}, \mathtt{Y})) = \sigma(\mathtt{lecturer}(\mathtt{COMP329}, \mathtt{Y}))$

Identifying applicable plans



Identifying applicable plans

- After having selected the relevant plans, we have to identify, among them, the applicable ones
- Applicable plans are those whose contexts is a logical consequence of the belief base
- **P** is a logical consequence of **Q** iff there exist a (most general unifier) σ such that $\sigma(P) = Q$
- Let's look at an example.

Identifying applicable plans

- Belief base

```
shape(box1,box)[source(percept)].
position(box1,coord(9,9))[source(percept)].
colour(box1,blue)[source(percept)].
shape(sphere2,sphere)[source(percept)].
position(sphere2,coord(7,7))[source(bob)].
colour(sphere2,red)[source(john)].
```

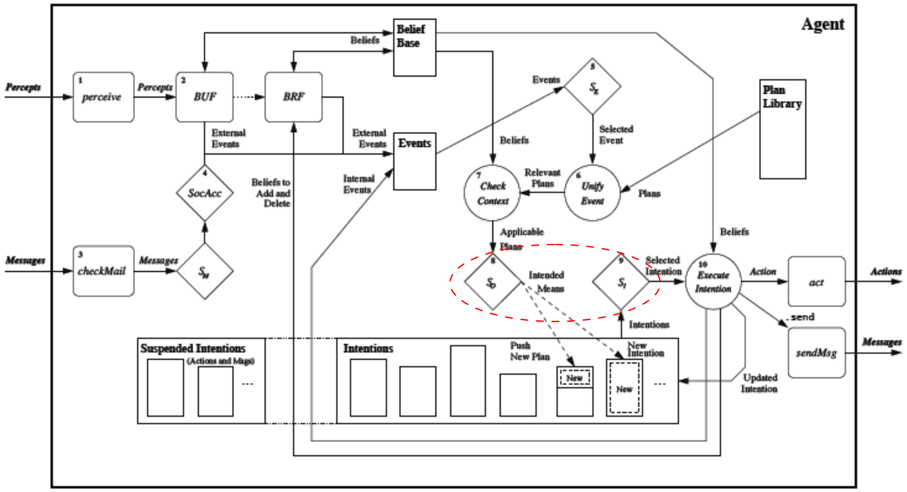
- Plans

```
+colour_(Object,Colour):
  shape(Object,box)
    & not position(Object,coord(0,0)) <- ...
+colour_(Object,Colour) :
  colour(OtherObj,red)[source(S)]
    & S/==percept &
    shape(OtherObj,Shape) &
    shape(Object,Shape) <- ...
```


Identifying applicable plans

- In this example, “a plan is a logical consequence of the belief base” just means that it is possible to coherently match (unify) elements of the belief base with a plan.
- However “logical consequence” allows the match to not only be with a fact in the belief base, but also with the result of applying a rule.

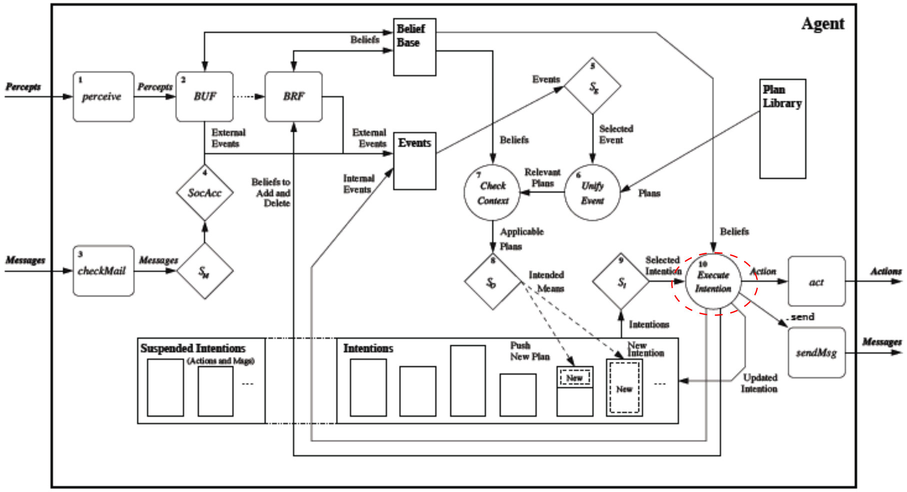
Selecting one plan and one intention



Selecting one plan and one intention

- Once the set of applicable plans has been determined, one among them has to be selected
- This is done by an option selection function .
  Its default setting works on a first-in first-out basis
- The selected plan is instantiated by the unification that determined it as applicable, and added to an intention stack , representing a single intention
- Several intentions (= stacks of partially instantiated plans) might be awaiting processing
- Again, a selection function (intention selection function) determines which intentions to process first
  Default: first-in first-out

Execute one step of an intention



Execute one step of an intention

- Suppose the selected event is:

  `<+b , T>`
- Then the selected plan:

  `[+b :  true <- !g ; a1 | ...  ]`

  is pushed onto the intention stack
- The interpreter then does intention selection. Let's assume that this pulls this same intention from the stack.
- The interpreter selects the first formula in the plan body:

  `!g`

  and pushes the rest of the intention back onto the stack.

Execute one step of an intention

- `!g` is a goal.
- Handling a goal involves creating the following event:

  `<+!g , [ +b :  true <- !g ; a1 ]>`
- Which then needs a plan.

Execute one step of an intention

- The system then repeats the previous couple of steps, selecting a plan for `+!g` and stacking the plan on the intention stack.
- Let's say that this plan:

  `[+!g :  true <- a2 | +b :  true <- !g ; a1 | ...  ]`

  is the one selected.
- Again it is pushed onto the intention stack, and a new intention selected.

Execute one step of an intention

- Assume this intention is:
  `[+!g :  true <- a2 | +b :  true <- !g ; a1 | ...  ]`
- This time the first step is an action `a2`.
- The agent just does this.
- The rest of the intention is pushed back onto the stack.

Execute one step of an intention

- Note that there are two different things happening.
- Executing goals removes one step from an existing intention,but pushes a new intention onto the stack.
  Stack grows.
- Executing an action removes one step from an existing intention
  Stack shrinks.
- With a LIFO intention stack we handle intentions in a recursive manner.
  - (The book says "FIFO", but I am pretty convinced it is a LIFO structure)
- A custom intention stack might prioritize intentions, for example by expected utility.

Plan failure

- Plans may fail for the following reasons:
  - The set of applicable plans turns out to be empty
  - An action fails
    i.e., no feedback reaches the agent about the successful execution of the action
  - A test goal fails
- If a plan for handling a goal achievement fails, Jason generates a goal-deletion event (and possibly drop the intention):
  `-!g`
- This event can be used by the programmer to specify further plans to handle the failure
- These can be as simple as:
  `-!g : true <- !g`

Summary

- This lecture focused on the structure of the Jason interpreter.
- It looked at the interpreter as a (sophisticated) model of a deliberation cycle.
- And it explained, in quite a lot of detail, all the main steps of the interpreter.