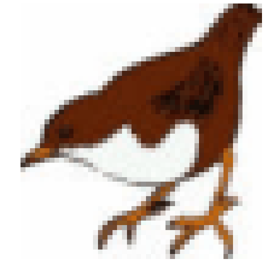
Building Spoken Dialogue Systems with DIPPER

DIPPER...



- Offers an architecture for prototyping spoken dialogue systems
- Is based on the Open Agent Architecture
- Has it own Dialogue Management Component, based on the information-state approach (Trindi)

Overview of this Talk

- The Dipper environment
  - Open Agent Architecture (OAA)
  - Agents and Solvables
  - Dialogue Management in Dipper
- The Information-state update approach
  - Information states
  - Update Language
- Comparison with TrindiKit
- Working with Dipper

1. The DIPPER environment

- How to build a dialogue system using and adapting off-the-shelf components that
  - need to interact with each other
  - are implemented in various programming languages
  - are running on various platforms?
- Examples:
  - Festival (C++), Nuance (C,C++,Java)
  - Parsing, Context Resolution (Prolog)
  - Dialogue Management (Prolog), O-Plan (Lisp)

The Open Agent Architecture

- Framework for integrating a community of heterogeneous software agents in a distributed environment
- Agents can be created in multiple programming languages on different platforms
- Agents can be spread across a computer network
- Agents can cooperate or compete on tasks in parallel

OAA Philosophy

- express requests in terms of *what is to be done* in terms of **solvables** without requiring specifying
  - who is to do the work
  - how it should be performed
- requester delegates control for meeting a goal with the **facilitator** (coordinating the activities of agents)
- develop components of application separately by wrapping them into **agents**

OAA Availability

- Developed by SRI AIC, freely available.
- Current Version OAA-2.1 (released Sept'01)
  - libraries for Java, C, C++, Prolog, and WebL
  - Solaris, Linux, and Windows 9x/NT
- OAA-1.0
  - more languages (Lisp, Basic, Delphi, Perl *etc.)*
  - SunOs 4.1.3, SGI IRIX
- OAA-2.1 Facilitator provides backward compatibility
  - OOA-1 and OAA-2 agents can co-exist
- Active community exists

OAA Agent Types

- *requester:* specifies goal to the facilitator, provides advice on how it should be met
- *providers:* register their capabilities with the facilitator, know what services they provide, understand limits of their ability to do so
- *facilitator:* maintains a list of provider agents and a set of general strategies for meeting goals

Prolog wrapper for requester

```
:- use_module(_, com_tcp, all).
:- use_module(oaa, all).

runtime_entry(start) :-
        com_Connect(parent, [], _Address),
        oaa:oaa_Register(parent, prolog_testagent2, [],[]),
        oaa:oaa_Ready(true).

:- runtime_entry(start).

% request service with:  oaa_Solve(test(A,B), Z).
```

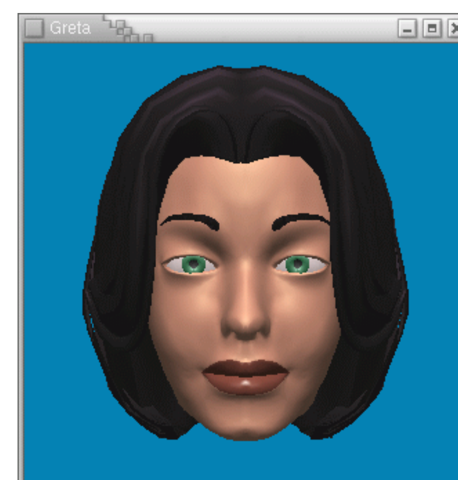
Prolog wrapper for provider

```
:- use_module(_, com_tcp, all).
:- use_module(oaa, all).

runtime_entry(start) :-
        com_Connect(parent, [], _Address),
        oaa:oaa_Register(parent, prolog_testagent1, [test(_A,_B)],[]),
        oaa:oaa_RegisterCallback(app_do_event, user:oaa_AppDoEvent),
        oaa:oaa_MainLoop(true).

oaa_AppDoEvent(test(A,B), Params) :-
        write(['I have been called with ', A, B]), nl,
        highly_complex_prolog_code(A,B).

highly_complex_prolog_code(A,B) :-
    A = 'a',
    B = 'b'.

:- runtime_entry(start).
```

Dipper: Input/Output Agents

NUANCE

- ASR: Dipper supports agent “wrappers” for **Nuance** 7.0 and 8.0 with solvables:
  - `apply_effects(+Effects)`
  - `recognize(+Grammar,+Time,-Result)`
- Synthesis: **Festival, rVoice, Greta**, with solvables:
  - `text2speech(+Text)`
  - `sable2speech(+SABLE)`
  - `play_apml(+APML)`



Dipper: Supporting Agents

- OAA comes itself with Gemini
  - parsing and generation
- Dipper provides further agents
  - DRT stuff (resolution, inference)
  - Theorem proving (SPASS, MACE)
  - Content planning (O-Plan)
  - X-10 Device control (Heyu)

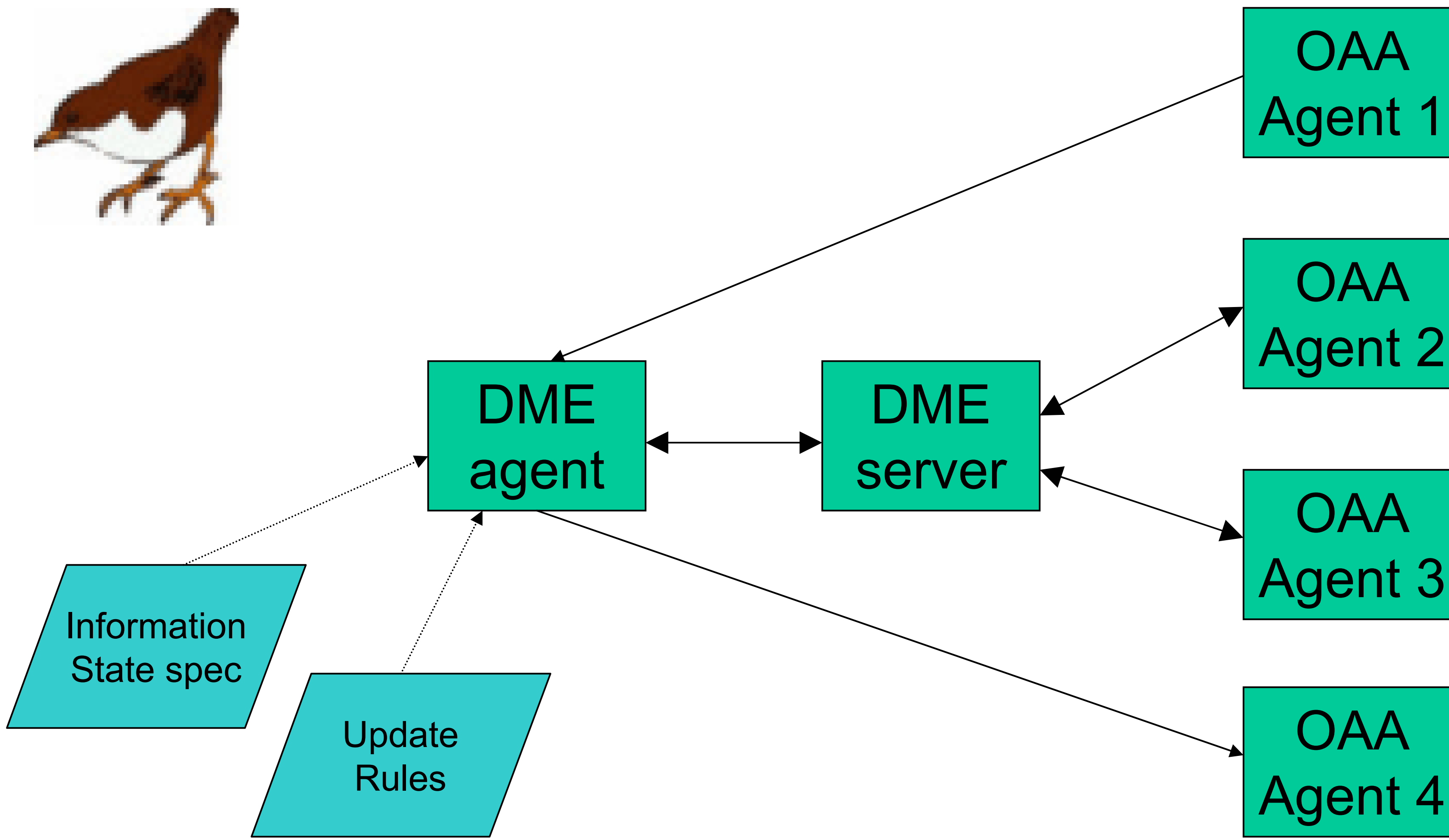
Dipper: Dialogue Management Agents

- Dialogue management forms the heart of a dialogue system:
  - Reading (multi-modal) input modalities
  - Updating the current state of the dialogue
  - Deciding what to do next
  - Generating output
- It is the most complex agent!
- Dipper implements dialogue management as two agents: the DME, and the DME server

The Dialogue Move Engine

- The DME agent, with solvables:
  - `check_conds(+Conditions)`
  - `apply_effects(+Effects)`
- The DME server mediates between the DME agents and other agents
  - `dme(+Call,+Effects)`
- Multiple threads possible

Dipper DME functionality



2. The Information-State Approach

- Some History
- The Information-state Approach
- Specifying Information States
- The Dipper Update Language
- A simple example

Some History

- Traditional approaches:
  - **Dialogue state** approaches (dialogue dynamics specified by a set of states and transitions modelling dialogue moves)
  - **Plan-based** approaches (used for more complex tasks showing flexible dialogue behaviour
- Information-state approaches combine the merits of both approaches

Information-state Approaches

- Declarative representation of dialogue modelling
- Components:
  - Specification of contents of the information state of the dialogue
  - Datatypes to structure information
  - A set of update rules
  - Control strategy for information state updates
- First implementation: TrindiKit
- Dipper builds on TrindiKit

Specifying Information States

- The information state "*represents the information necessary to distinguish it from other dialogues, representing the cumulative additions from previous actions in the dialogue, and motivating further action*" (Traum et al., 1999)
- Compare: mental model, discourse context, state of affairs, conversational score, etc.
- Dipper uses TrindiKit technology representing information states

Example: Information State Definition

Datatypes: record, stack, queue, atomic, drs

```
is:record([grammar:atomic,
           input:queue(atomic),
           sem:stack(record([int:atomic,
                             context:drs]))])
```

Information State based on Ginzburg's QUD (Godis)

- `Private:`
  - `Bel`: set of propositions (according to system)
  - `Agenda`: stack of actions (short-term intentions)
  - `Plan`: stack of actions (long-term dialogue goals)
  - `Tmp`: copy of Shared
- `Shared`:
  - `Bel`: set of propositions (shared by participants)
  - `QUD`: stack of questions under discussion
  - `LM`: latest move (speaker, move, content)

The Dipper Update Language

- Update Rules have 3 components
  - **Name** (identifier)
  - **Conditions** (a set of ‘tests’ on the current information state)
  - **Effects** (an ordered set of operations on the information state, resulting in a new state)
- Conditions and effects are defined by the Dipper Update Language

Standard vs Anchored Terms

- Standard Terms: basic definitions of the datatypes (constants, stacks, queues, records)
- Special term: `is`, referring to the complete information state
- Anchored Terms
  - `is`, T^F, first(T), last(T), top(T), member(T)

Example: Anchored Terms

- Information State (s)

```
is: grammar: '.Yesno'
     input: <>
     sem: < int: model(…)
            context: drs([X,Y],…) >
```

- Reference: [[.]]s
  - [[is^grammar]]$_s$ = '.Yesno'
  - [[grammar]]$_s$ = grammar
  - [[top(is^sem)^context]]$_s$ = drs([X,Y],…)
  - [[top(sem)^context]]$_s$ = *undefined*

Conditions and Effects

- Conditions
  - T1=T2, T1\=T2
  - `empty`(T1), `non_empty`(T1)
- Effects (T1 anchored)
  - `assign`(T1,T2), `clear`(T1), `pop`(T1), `push`(T1,T2), `dequeue`(T1), `enqueue`(T1,T2)
  - `solve`(x,S(…,Ti,…),Effects)

A Simple Example: Parrot

- We will use the following information state structure:

```
is:record([input:queue(atomic),
           listening:atomic,
           output:queue(atomic)])
```

- Four agents:
  – ASR, SYN, the DME agent and the DME server

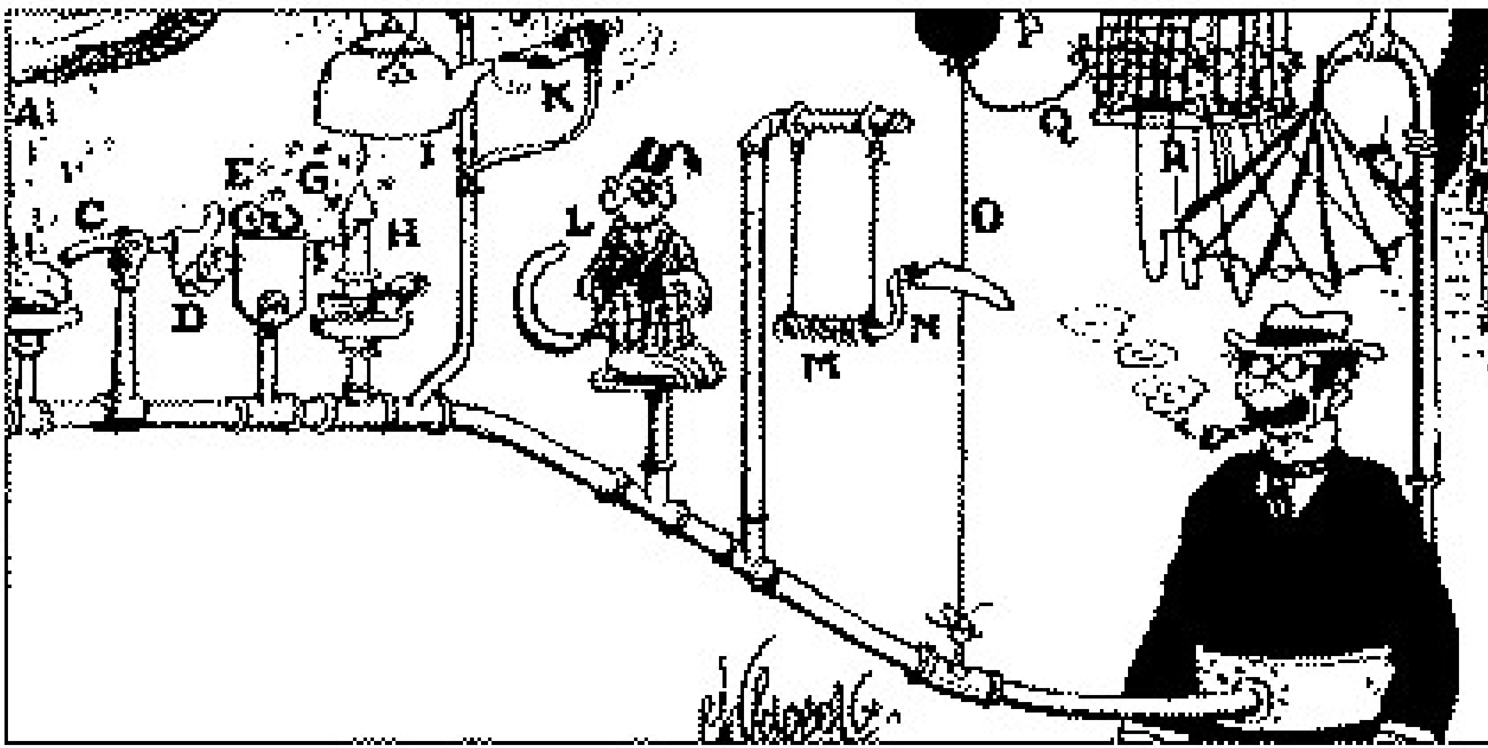
Update Rules for Parrot

```
urule(timeout,
      [first(is^input)=timeout],
      [dequeue(is^input)]).

urule(process,
      [non_empty(is^input)],
      [enqueue(is^output,first(is^input),
       dequeue(is^input)]).

urule(synthesise,
      [non_empty(is^output)],
      [solve(_,text2speech(first(is^output)),[]),
       dequeue(is^output)]).

urule(recognise,
      [is^listening=no],
      [solve(X,recognise('.Gram',10),
           [enqueue(is^input,X),assign(is^listening,no)]),
       assign(is^listening,yes)]).
```

3. Comparison with TrindiKit

- TrindiKit (Larsson, Berman, Bos, Grönqvist, Ljunglöf and Traum 1999)
  - first implementation of information-state approach
- Complexity of TrindiKit obscures what should be a simple and transparent operation:
  - updating information state with declarative update rules

“Rube Goldberg” Machine



Dipper vs. TrindiKit: Control

- Dipper
  - Information state
  - update rules
- TrindiKit
  - (Typed) update rules and selection rules
  - Update algorithms (DME-ADL)
  - Control algorithms
  - TIS (IS + MIVs + RIVs)

Dipper vs. TrindiKit OAA Integration

- Dipper
  - OAA solvables in effects of update rules
  - Allows easy integration of components without touching the dialogue engine
  - The DME is just one of the agents
- TrindiKit
  - No OAA solvables in update rules,
  - Module Interface variables
  - TrindiKit is an architecture for “everything”

Dipper Update Algorithm

```
WHILE running
        deal with OAA-events;
        IF there is an applicable rule
        THEN apply its effects
ENDWHILE
```

Dipper vs. TrindiKit: Use of Variables

- DIPPER update language is essentially variable-free (reference with anchors)
- TrindiKit relies on Prolog variable unification (reference with variables)
- Example 1:
  - Dipper: [push(is^b,top(is^a)),pop(is^a)]
  - TrindiKit: [is::fst(a,X),is::pop(a),is::push(b,X)]
- Example 2:
  - Dipper: [assign(top(is^sem)^int,m)]
  - TrindiKit: [is::fst(sem,X),X::set(int,m)]

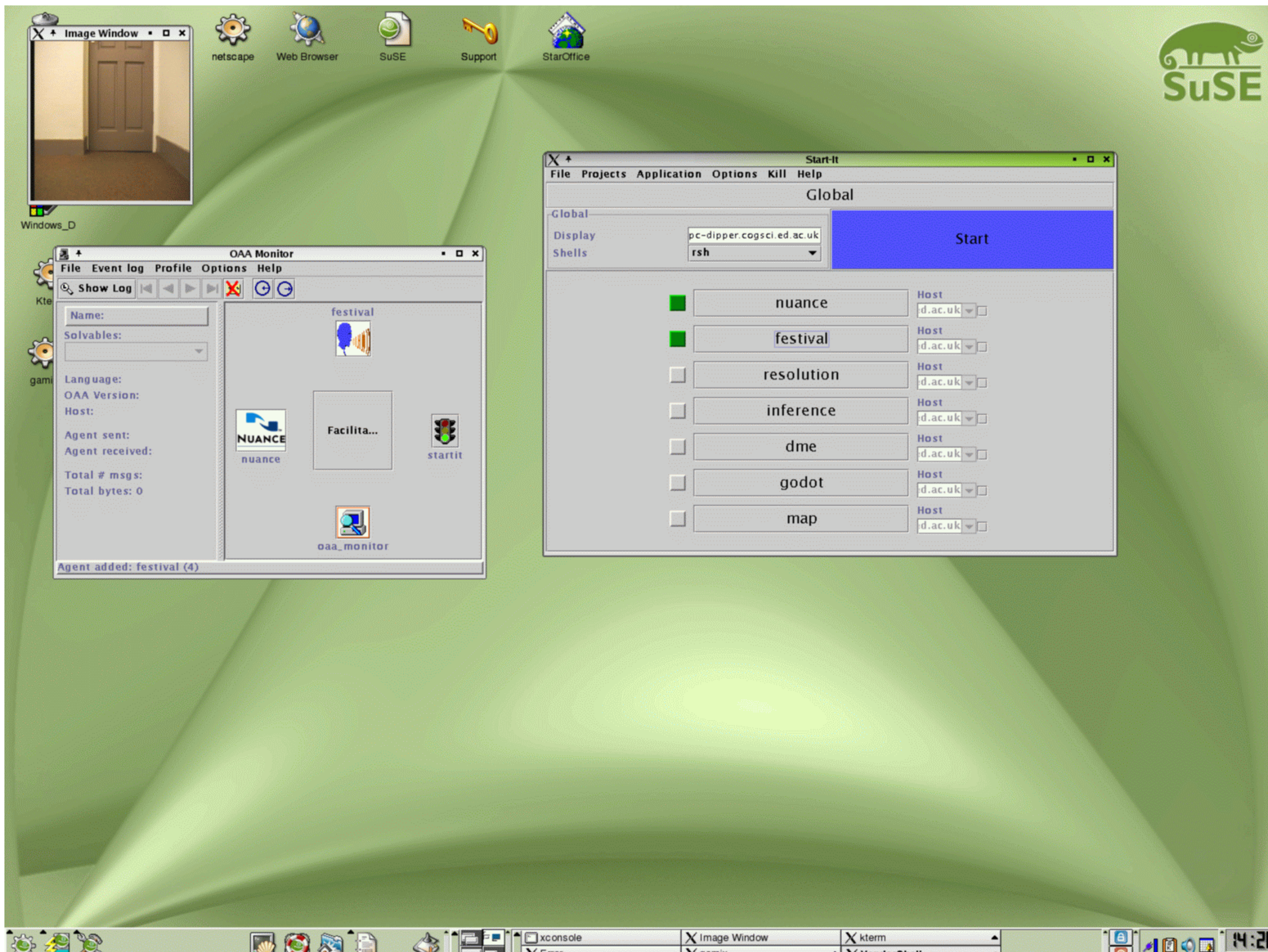
4. Working with DIPPER

- Prototyping
  - How to build and run a DIPPER application
  - The startit.sh and monitor.sh
- Debugging
  - Testing and debugging of information-state approaches can be difficult
- DIPPER prototypes

How to build and run a DIPPER application?

- Set up your machine for using OAA (and Nuance)
- Decide which components you want to use and specify an OAA config file
- Specify information state and update rules
- Start the OAA facilitator (fac.sh) and the OAA application manager (startit.sh)

OAA tools (startit.sh and monitor.sh)



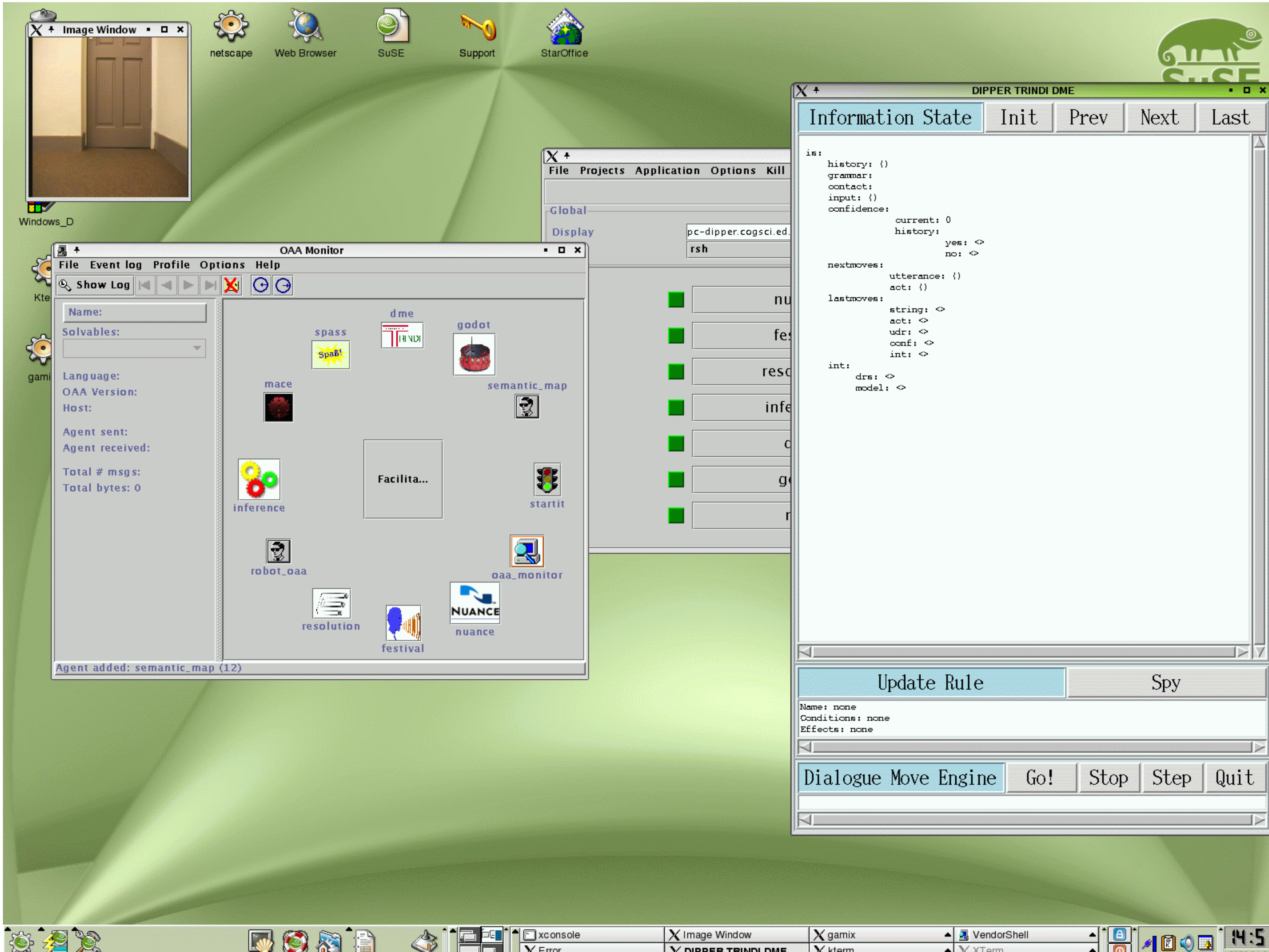
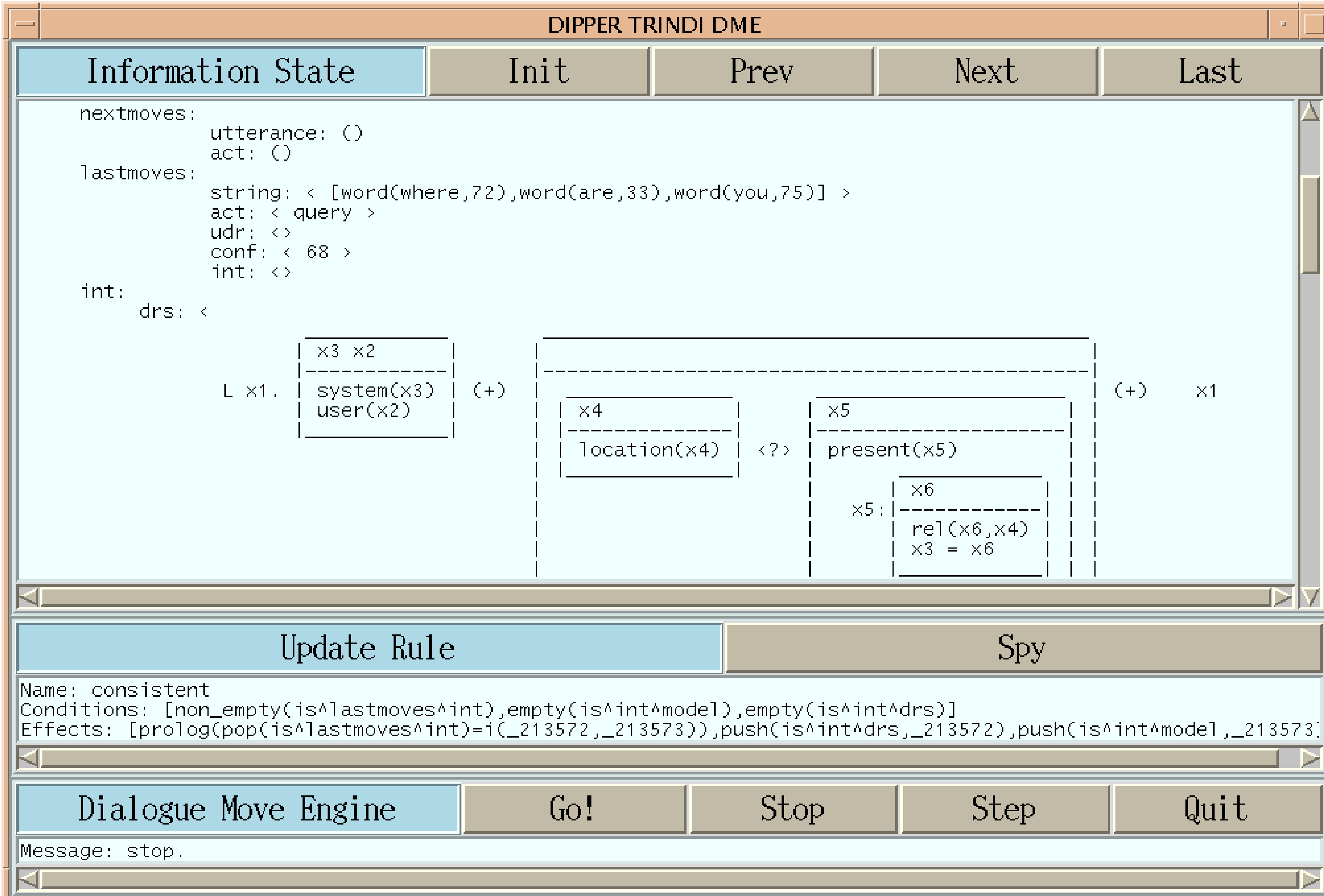


Image Window
netscape
Web Browser
SuSE
Support
StarOffice
Windows_D
DIPPER TRINDI DME
Information State
Init
Prev
Next
Last
is:
history: ⟨⟩
grammar:
contact:
input: ⟨⟩
confidence:
current: 0
history:
yes: <>
no: <>
nextmoves:
utterance: ⟨⟩
act: ⟨⟩
lastmoves:
string: <>
act: <>
udr: <>
conf: <>
int: <>
int:
drs: <>
model: <>
Update Rule
Spy
Name: none
Conditions: none
Effects: none
Dialogue Move Engine
Go!
Stop
Step
Quit
File Projects Application Options Kill
Global
Display
pc-dipper.cogsci.ed.
rsh
OAA Monitor
File Event log Profile Options Help
Show Log
Name:
Solvables:
Language:
OAA Version:
Host:
Agent sent:
Agent received:
Total # msgs:
Total bytes: 0
spass
dme
godot
mace
semantic_map
inference
Facilita...
startit
robot_oaa
oaa_monitor
resolution
festival
nuance
Agent added: semantic_map (12)
xconsole
Error
Image Window
DIPPER TRINDI DME
gamix
kterm
VendorShell
XTerm
14:51
2003-01-21

The DIPPER GUI



Dipper Prototypes

- D’Homme (home automation)
- IBL (route explanation to mobile robot)
- Godot (our own robot in the basement)
- Magicster (believable agents)

- Dipper Resources:

  `http://www.ltg.ed.ac.uk/dipper`