Design of Multi Agent Systems

Assoc. Prof. Lin Padgham

School of Computer Science and Information Technology

RMIT University

Tutorial at Net.ObjectDays (NODe 2002)

Monday 7th October 2002
Erfurt, Germany

This page intentionally blank

Contents

Slides

Exercises

Prometheus: A Methodology for Developing Intelligent Agents [1]

Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents [2]

Note: the version in these notes is a DRAFT.

Simplifying the Develoment of Intelligent Agents [3]

Forms

Resources

[1] Lin Padgham and Michael Winikoff. *Prometheus: A Methodology for Developing Intelligent Agents.* In proceedings of the the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS'02, July 2002.

[2] Lin Padgham and Michael Winikoff. *Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents.* To appear in proceedings of the workshop on agent-oriented methodologies at OOPSLA'02, 4 November 2002.

[3] Michael Winikoff, Lin Padgham, and James Harland. *Simplifying the Development of Intelligent Agents.* In AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence. LNAI 2256, pages 557-568, Adelaide, December 2001.

What is an Agent?

"encapsulated computer system, situated in some environment, and capable of flexible autonomous action in that environment in order to meet its design objectives" (Wooldridge)

- ***control over internal state and over own behavior***
- ***experiences environment through sensors and acts through effectors***
- ***reactive: respond in timely fashion to environmental change***
- ***proactive: act in anticipation of future goals***

What "agent" can mean

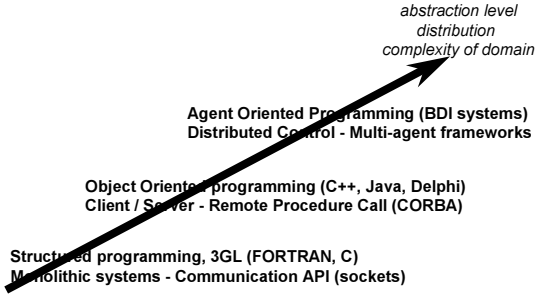
- A term used in many ways
- A range of software varying in capability, from simple to highly sophisticated
  - procedural "wizards" found in popular desktop applications
  - information agents for location and retrieval of information from multiple, dispersed data sources
  - intelligent (*intentional*) agents capable of performing simple rational reasoning tasks and balancing goal-directed and reactive behaviour

What agents are not

- Wizards
- Scripting languages
- Programming languages
- Neural Networks
- Pure rule-based systems
- Objects

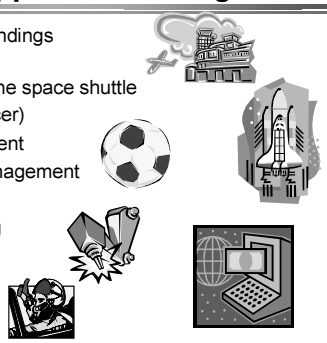


Agent oriented software



Some Applications of Agents

- Sequencing aircraft landings at Sydney airport
- Diagnosing faults on the space shuttle
- Robocup (robotic soccer)
- Information management
- Business process management
- eCommerce
- Holonic manufacturing
- Defence simulation
- …



Parent technologies

- artificial intelligence
- software engineering
- distributed systems
- organisational science
- databases
- economics
- game theory
- artificial life



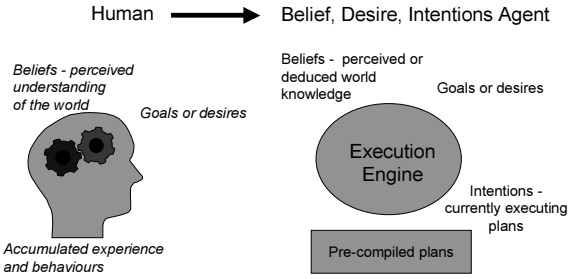
Types of agents

- information agents
- mobile agents
- collaborating agents
- negotiating agents
- agents with personality
- intentional agents

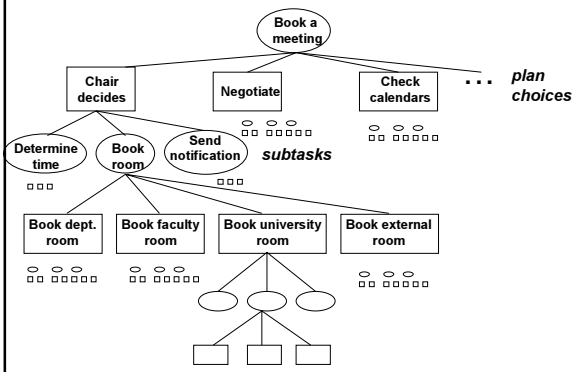
more generally …

- a paradigm for distributed computation and decision making
- in open, heterogeneous dynamic environments
- with multi-threaded, asynchronous message-passing processing

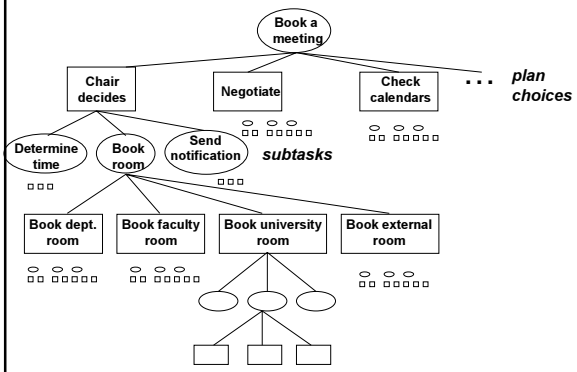
The BDI Agent Model



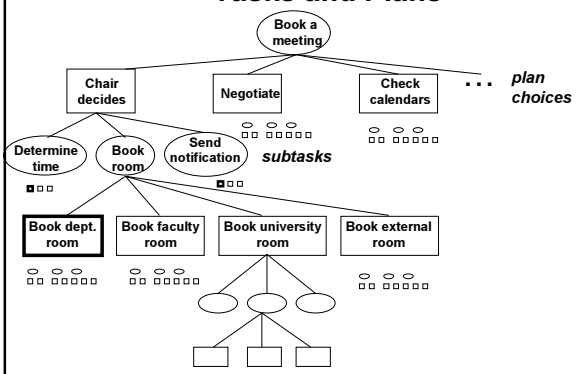
Tasks and Plans



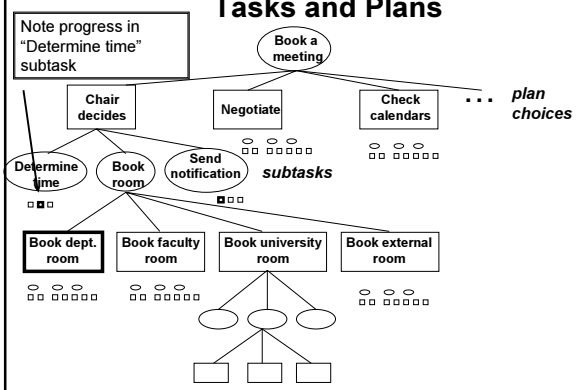
Tasks and Plans



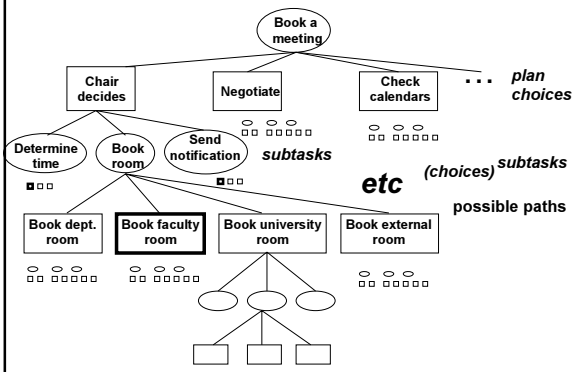
Tasks and Plans



Tasks and Plans



Tasks and Plans



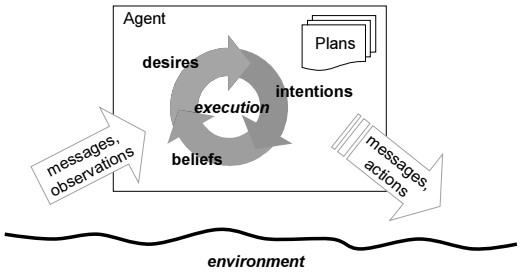
Space of possible executions

- Grows very rapidly as you increase the number of subtasks and the number of ways of doing a subtask.
- $((\text{choices})^{\text{subtasks}})^{\text{depth}}$
- With 3 subtasks/plan, 3 choices/(sub)task and 2 subtasks between top level goal and actions we have 729 possible ways of achieving a given goal X!!

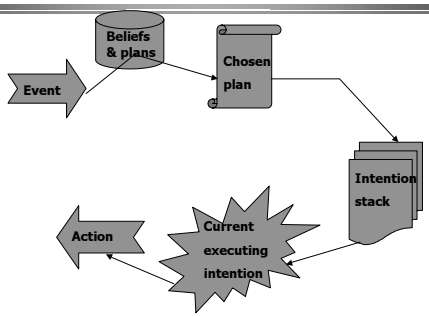
Maximising power of BDI

- Use subgoals wherever possible (rather than actions or in-line code)
- Write many simple alternative plans, rather than more complex comprehensive plans.
- Design (and code) for alternative ways to do things, even if only one way initially (prefer subgoal and single plan to in-line code).

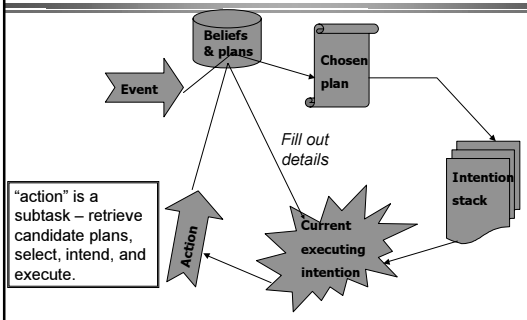
BDI Execution Concepts



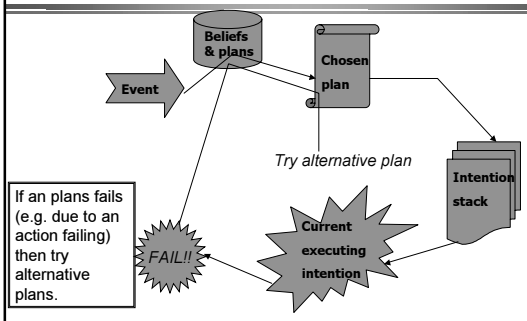
BDI control structure



BDI control structure



BDI control structure



Goal persistence

- If a particular plan fails, the agent still maintains the goal and tries to achieve it using other plans.
- If a subtask fails, the execution engine looks for another plan to try to achieve that subtask. It won't retry the same plan.
- If there are no more plans to try for a subtask, then the parent plan fails - and the execution engine looks for an alternative to it.

Pragmatic plan based agents

- Use pre-compiled plans - much faster than first principles planning
- Sensitive to environmental changes - chooses each plan/sub-plan as needed.
- If it fails due to environment change while executing, then tries a new plan for new environment
- Recognize that world is dynamic
- Huge space of possible paths available, without explicit coding.
- Development systems include JACK, PRS, JAM, ...

Programming advantages

- Use of high level concepts and constructs
- Modularity
  - plans are encapsulated pieces
  - beliefs can be organised in separate knowledge bases
- Powerful execution model
  - chooses plan based on current situation
  - aborts or suspends if situation changes
  - orders actions according to priorities
  - goal achievement semantics
- Largely declarative in style

Design of Intelligent Agent Systems

- Development of agent systems requires a new way of thinking about design
- Goals, agents, plans, rather than objects and methods
- Prometheus methodology has grown out of a need to find a way to assist students and industry developers in building (BDI) agent systems.

Strengths of Prometheus

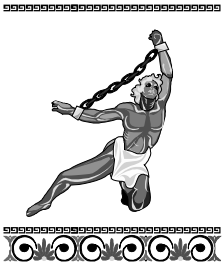
- Support for **intelligent** agents (particularly in detailed design)
- **Hierarchical** approach makes it suitable for large systems
- Based on **pedagogical and industry** experience
- Support for **full life cycle** is being developed: Specification, Architectural Design, Detailed Design, Implementation, Testing and Debugging, Modification

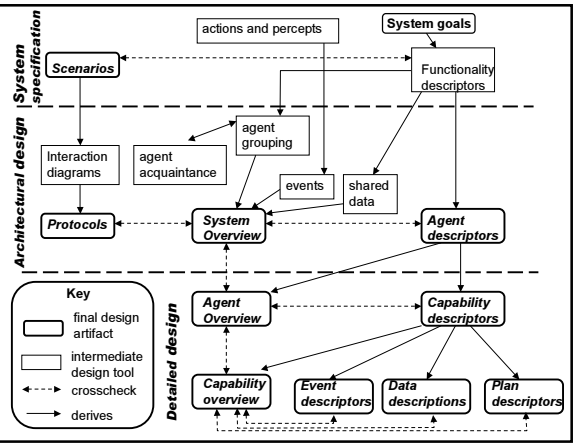
Strengths of Prometheus ctd.

- **Structured design artifacts** that support **automated design checks**.
- Defined **process steps** for producing artifacts.
- Clear **linkages between artifacts** of different phases.
- Ability to **automate support** for production of design and design document.

"Prometheus"?

Prometheus was the wisest Titan. His name means "forethought" and he was able to foretell the future. Prometheus is known as the protector and benefactor of man. He gave mankind a number of gifts including fire. (*http://www.greekmythology.com/*)





Design Methodology

SYSTEM SPECIFICATION

1. Determine external interfaces - actions, percepts, permanent data
2. Determine system goals, functionalities and scenarios

ARCHITECTURAL DESIGN

3. Define Agents, Incidents, and Interactions. Also shared data structures.

DETAILED DESIGN

4. Define Capabilities, Plans, Belief structures, Events

Design Criteria

- Modularity
  - [system] = [module1] + [module2] + [module3] + ...
- Cohesion
  - "Module X is ..."
- Coupling
  - "Modules X and Y interact about ..."
- Interfaces
  - "Module X provides ... and needs ..."

Focus on *goals* and *functionalities*

- Identify system goals - main and subsidiary
- Identify functionalities that will achieve these goals - these should be narrow and definable in 1-2 sentences.
- Identify permanent data stores (i.e. external)
- Prepare Functionality Descriptors
- Define Scenarios (and variations)
- Ensure all goals covered by scenarios

Example Goals and Functionalities: Air traffic control

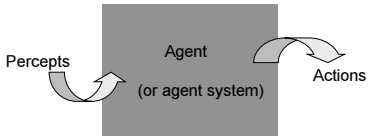
Goals:

1) Land planes safely
2) Have as little delay as possible
3) Keep planes separated
4) Land with adequate fuel reserves

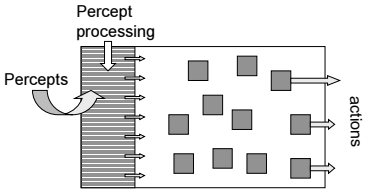
Functionalities

schedule manager, (2)
separation manager, (3)
fuel watcher, (4)
weather observer, (1)
runway watcher, (3, 2, 1)
…

Agents are situated



Define the interface(s) with the environment



Percepts

- May need to process percepts to determine whether something of interest has occurred - an event that is of interest to the agent.
- Look for changes between current and previous percepts. Or between believed state of world and percept.
- E.g. If a ball is in a different position, compared to last percept, we may want to generate a ball-moved incident. Or if we look at a position we had believed the ball in (based on messages, or past vision, or …) and its not there, then we may want to generate a ball lost incident - or update our beliefs.

Actions

- Recall that agents are **situated** - they obtain ongoing information from the environment, and they do things which affect the environment.
- The things which the agent does which affect the environment we call **actions**.
- Actions may be simple and direct, or they may be complex - particularly in physical systems actions may involve complex code interacting with mechanical devices.

Data used and produced by functionalities

- What information does each functionality need?
- What information does each functionality produce?
- For information needed - where does it come from? Another functionality? A percept? Initialisation information?
- What external data stores seem to be required?

Functionality Descriptors

Detailed description of each functionality :

NAME: short descriptor

DESCRIPTION: natural language description, reasonable detail If more than 3 sentences, consider breaking down.

PERCEPTS/TRIGGERS: what activates this functionality.

ACTIONS: actions on the external environment this functionality is responsible for.

MESSAGES: messages to other functionality (often triggers)

INFORMATION USED: what data does this functionality need

INFORMATION PRODUCED: what data is produced

INTERACTIONS: what other functionalities does this functionality interact with

Scenarios

- Scenarios describe processes within the system to achieve the system goals (and deal with problems)
- Similar to O-O use-case scenarios - a series of steps
- No attempt at complete coverage, but should give detailed "flavour" of system
- Easy for clients (or others) to understand

Scenarios

- Series of steps where each step is:
  - INCIDENT/ PERCEPT: name (functionality)
  - message name: (fn1 -> fn2)
  - ACTION: name (functionality)
  - activity name (functionality)
  - [scenario (id)]
- Data used / produced can also be attached to each step

Scenarios

- Each system goal should be represented in at least one scenario
- Scenarios typically involve multiple functionalities
- Scenarios show a particular instance of system execution - no branching, etc.
- Alternatives are described by separate scenarios, or by variations (annotation describing the variation)
- Scenarios may contain steps which are themselves scenarios - these should be indicated
- Each step linked to functionalities and to data used/produced

Scenario Descriptors

- Scenario identification - name and number
- Description - short natural language
- Context in which scenario occurs
- Scenario steps - with functionalities and data
- Scenario variations - may be several. If variation significant, make separate scenario

Example Scenario

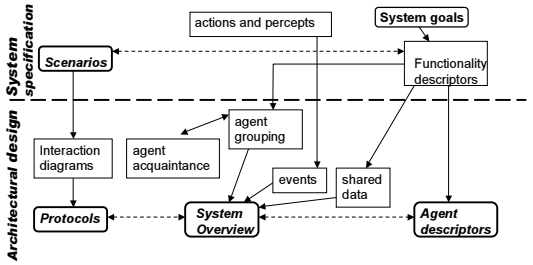
- INCIDENT New population threatened (civilian safety watcher)
- evacuation request (civ. Safety watch -> resource alloc)
- deploy vehicles (resource alloc. -> police deployment)
- loop:
  - get safe route (police deployment -> route planner)
  - route (route planner -> police deployment)
  - request police move (police deployment ->comms)
  - ACTION tell move-to (comms)
  - inform roads used (police deployment -> road safety)
- do evacuation [from,to] (civilian safety -> police deployment)
- get safe route (police deployment -> route planner)
- continued over…

example continued

- Route (route planner -> police deployment)
- inform roads used (police deployment -> road safety
- loop:
  - request police move people (police deployment -> comms)
  - ACTION move-people-to-via (comms)
  - PERCEPT/INCIDENT request next job (comms)
  - job request (comms -> police deployment)
- evacuation completed (police deployment -> civilian safety)
- ACTION update civilian information (civilian safety)

Architectural Design Phase

3. Define Agents, Interactions and shared Data Structures.



Grouping of Functionalities into Agents

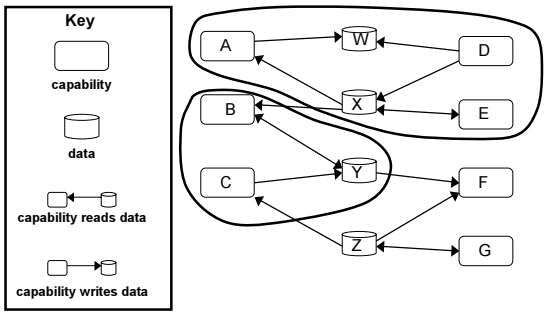
Reasons for:

- functionalities related - naturally belong together
- functionalities use the same data
- functionalities interact a lot

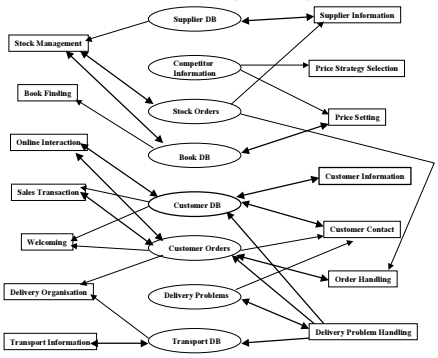
Reasons against:

- functionalities unrelated - a non-cohesive agent
- functionalities need to reside on different hardware platforms
- Differing cardinality - e.g. 1/system vs 1/customer

Data Coupling Diagrams



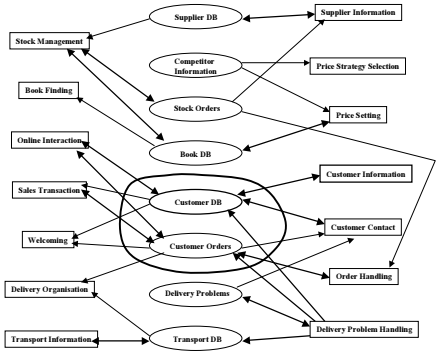
Data Coupling Diagram



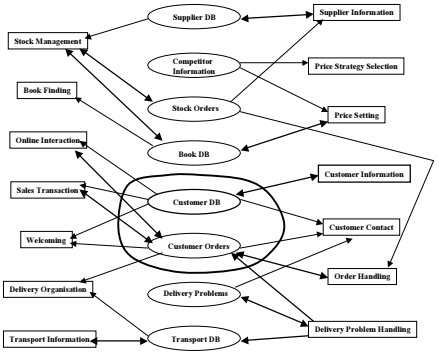
Notes

- Red bold arrows denote writing
- We want to avoid having multiple agents writing the same data, so we select one functionality (Customer Information) that will write to the Customer DB.
- All other functionalities that currently write will ask the Customer Information functionality to make the changes on their behalf
- The next two slides show this change

Data Coupling Diagram



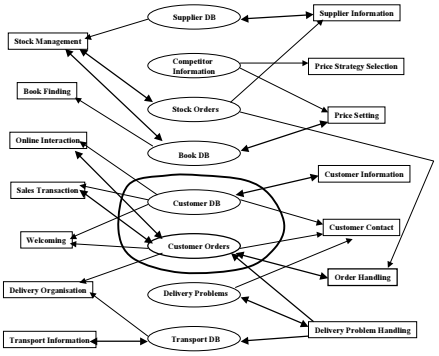
Data Coupling Diagram



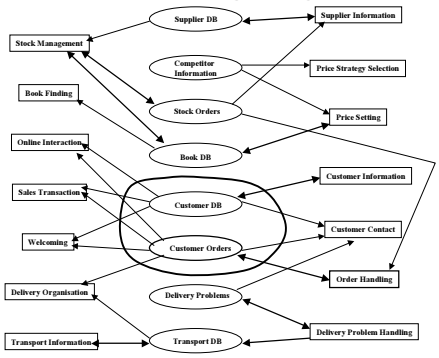
Notes

- Similarly, we select the Order Handling functionality to be the only functionality that writes to the Customer Orders.
- All other functionalities that currently write will ask the Order Handling functionality to make the changes on their behalf
- The next two slides show this change

Data Coupling Diagram



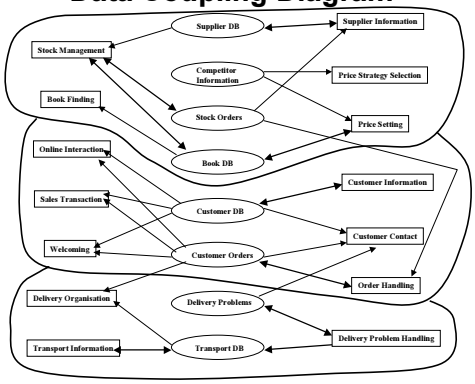
Data Coupling Diagram



Notes

- The next few slides show a number of alternative designs.
- We begin by identifying three agent types

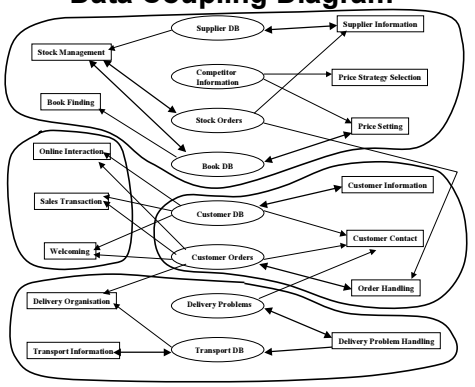
Data Coupling Diagram



Notes

- The design on the previous slide has a problem:
  - We want to have an agent for each active customer that encompasses the functionalities of welcoming, sales transaction, and online interaction.
  - However, we want to have a single agent in the system with the functionalities customer information, customer contact, and order handling.
- This forces us to split the middle agent typeinto two agent types.
- The next slide shows this change

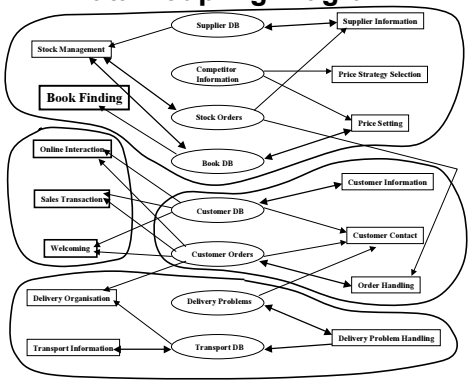
Data Coupling Diagram



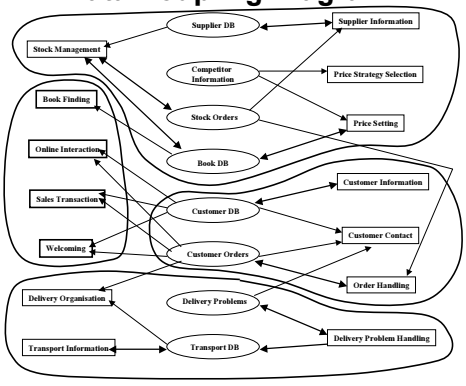
Notes

- We now investigate alternative designs.
- Firstly, the Book Finding functionality could be moved to be in the per-customer agent. This is shown in the next two slides.
- Secondly, the Order Handling functionality could be moved to be in the delivery agent type (the bottom grouping). This is shown in the subsequent two slides.
- Note that when moving Order Handling, we also need to move the data that it writes.

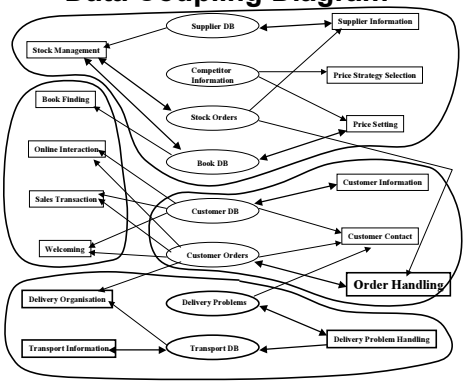
Data Coupling Diagram



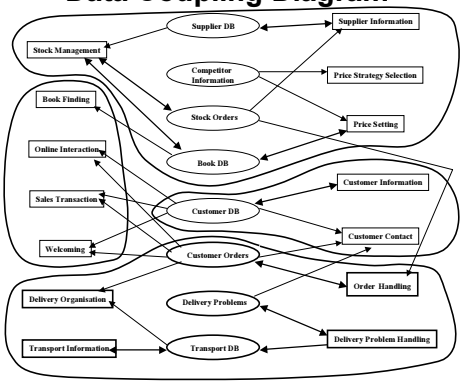
Data Coupling Diagram



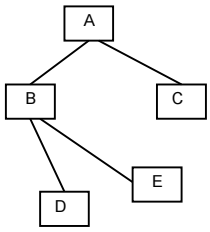
Data Coupling Diagram

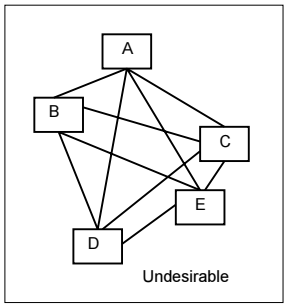


Data Coupling Diagram



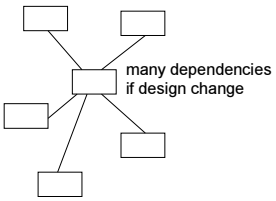
Agent Acquaintance Diagrams

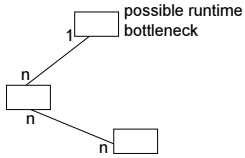




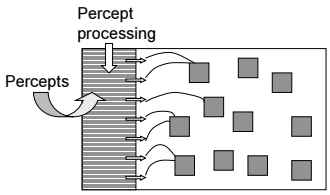
Agent Acquaintance Diagrams

Bottlenecks must also be considered



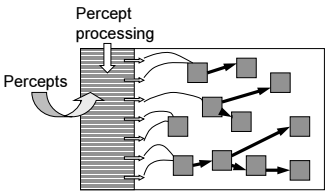


Link Percepts to Agents



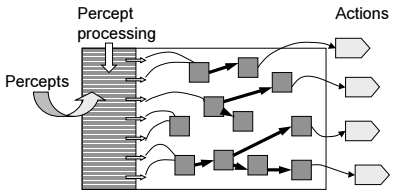
We need to determine which events are handled by which agents

Messages between agents



And what conversations or message protocols there are between agents

Actions by agents



What agents perform what actions on the environment

Percept and Action Descriptors

Percept	Action
• Name • Description • Source (*sensor etc.)* • Structure (*form of data*) • Events generated • Auxiliary data (*for generating events*)	• Name • Description • Parameters (*affecting performance)* • Temporality (*instantaneous, durational*) • Failure detection • Partial change • Side effects

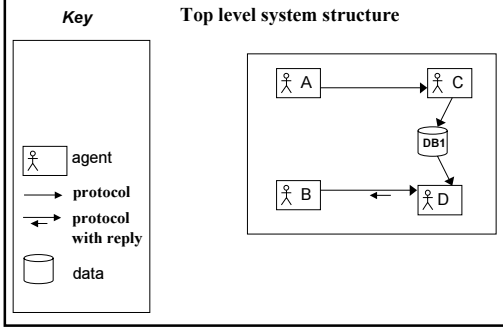
Agent Descriptors

- Name
- Description
- Incidents responded to
- Actions executed
- Messages sent - from which agent, what content
- Messages received - to which agent, what content
- Data used - external or shared internal
- Data produced - external or shared internal

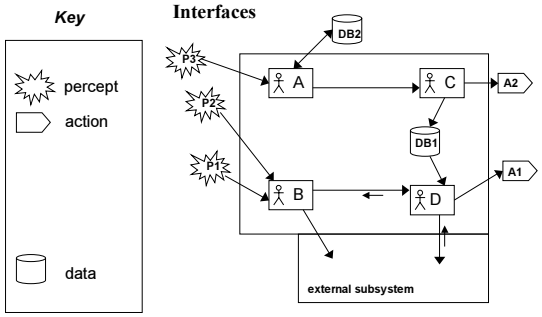
Protocol Descriptors

- Name and Description
- Information carried
- Initiating agent
- Other agents
- Related scenario(s)

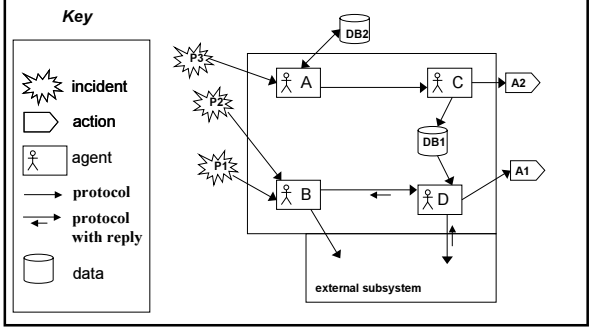
System Overview Diagram



System Overview Diagram



System Overview Diagram

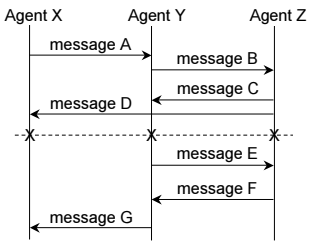


Interaction Diagram

- Purpose: design multi-agent co-operation activity
- Based on scenarios (and have same limitations)
- Between agents rather than between functionalities
- Can be used for determining interaction protocols, via generalisation

Interaction Diagram Example

- Agents
- Messages
- Time
- Single scenario
- Decision point



Scenario to interaction diagram

PERCEPT New population threatened (civilian safety watcher)
evacuation request (civ. Safety watch -> resource alloc)
deploy vehicles (resource alloc. -> police deployment)
for each car do:
    get safe route (police deployment -> route planner)
    route (route planner -> police deployment)
    request police move (police deployment ->comms)
    ACTION tell move-to (comms)
    inform roads used (police deployment -> road safety)
do evacuation [from,to] (civilian safety -> police deployment)
get safe route (police deployment -> route planner)
Route (route planner -> police deployment)
inform roads used (police deployment -> road safety
for each car do:
    request police move people (police deployment -> comms)
    ACTION move-people-to-via (comms)
    PERCEPT request next job (comms)
    job request (comms -> police deployment)
evacuation completed (police deployment -> civilian safety)
ACTION update civilian information (civilian safety)

Scenario to interaction diagram

PERCEPT New population threatened (civilian safety watcher)
evacuation request (civ. Safety watch -> resource alloc)
deploy vehicles (resource alloc. -> police deployment)
for each car do:
    get safe route (police deployment -> route planner)
    route (route planner -> police deployment)
    request police move (police deployment ->comms)
    ACTION tell move-to (comms)
    inform roads used (police deployment -> road safety)
do evacuation [from,to] (civilian safety -> police deployment)
get safe route (police deployment -> route planner)
Route (route planner -> police deployment)
inform roads used (police deployment -> road safety
for each car do:
    request police move people (police deployment -> comms)
    ACTION move-people-to-via (comms)
    PERCEPT request next job (comms)
    job request (comms -> police deployment)
evacuation completed (police deployment -> civilian safety)
ACTION update civilian information (civilian safety)

Civilian Monitor CM
Safety watch,
Evacuation priority

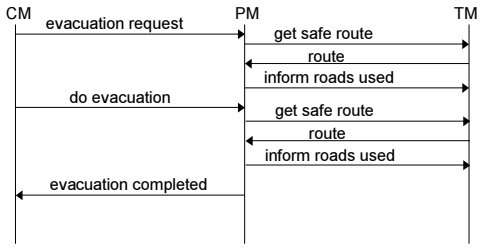
Terrain Manager TM
Map manager,
Road safety,
Route planning

Police Manager PM
Resource allocations
Comms,
Police deployment ,
Police locations

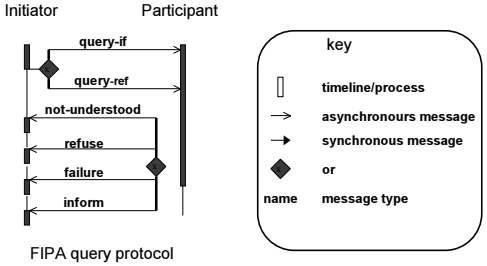
Scenario to interaction diagram

PERCEPT New population threatened (civilian safety watcher)
evacuation request (civ. Safety watch -> resource alloc) — CM ----------->PM
deploy vehicles (resource alloc. -> police deployment) — internal to agent
for each car do:
    get safe route (police deployment -> route planner) — PM ----------->TM
    route (route planner -> police deployment) — PM <-----------TM
    request police move (police deployment ->comms) — internal to agent
    ACTION tell move-to (comms)
    inform roads used (police deployment -> road safety) — PM ----------->TM
do evacuation [from,to] (civilian safety -> police deployment) — CM ----------->PM
get safe route (police deployment -> route planner) — PM ----------->TM
Route (route planner -> police deployment) — PM <-----------TM
inform roads used (police deployment -> road safety — PM ----------->TM
for each car do:
    request police move people (police deployment -> comms) — internal to agent
    ACTION move-people-to-via (comms)
    PERCEPT request next job (comms)
    job request (comms -> police deployment) — internal to agent
evacuation completed (police deployment -> civilian safety) — CM ----------->PM
ACTION update civilian information (civilian safety)

Interaction diagram



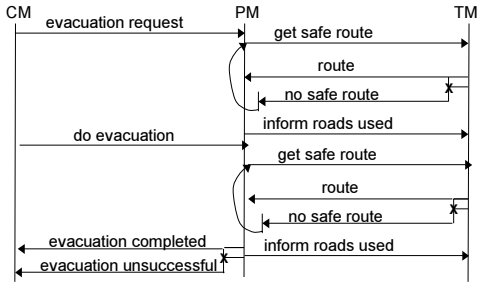
Generalise to Protocols

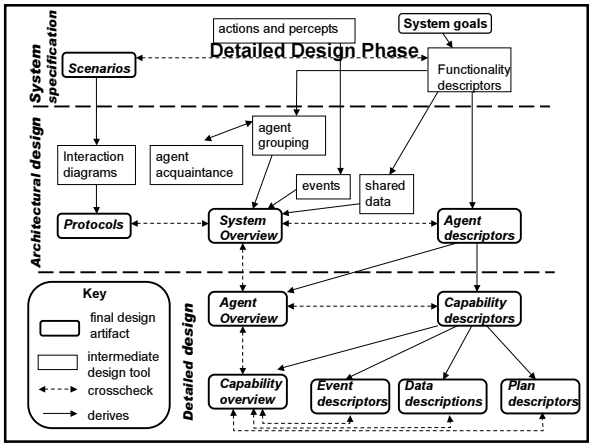


FIPA query protocol

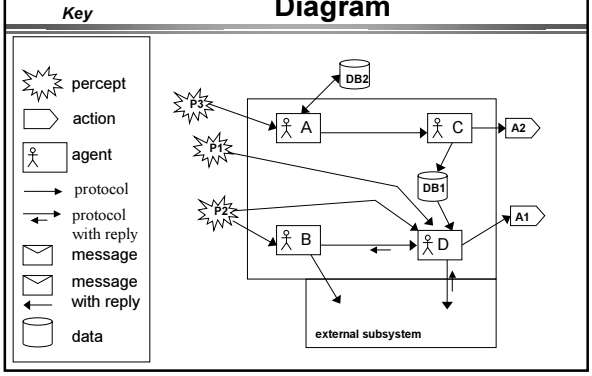
Papers with further information on possible syntax etc. available at www.auml.org

Interaction diagram to protocol

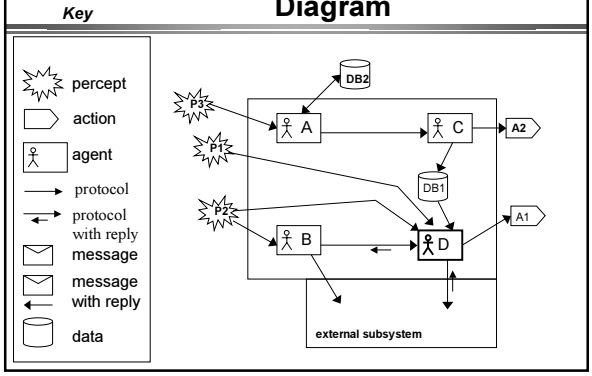




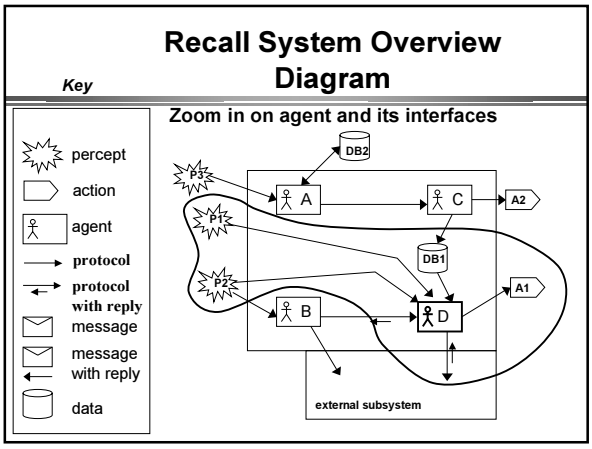
Recall System Overview Diagram



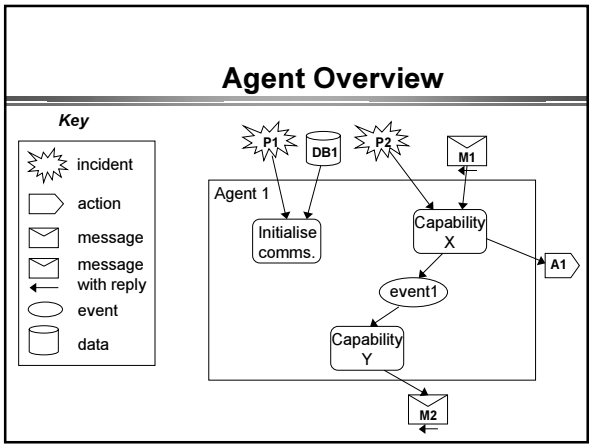
Recall System Overview Diagram



Recall System Overview Diagram



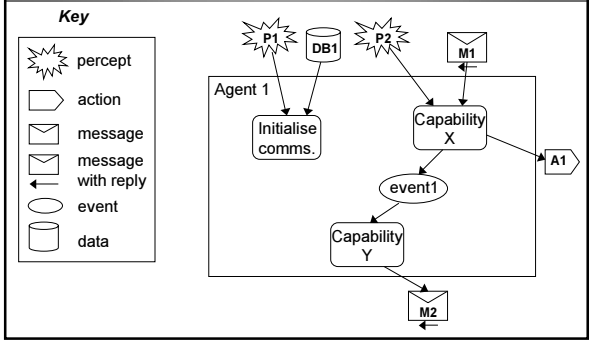
Agent Overview



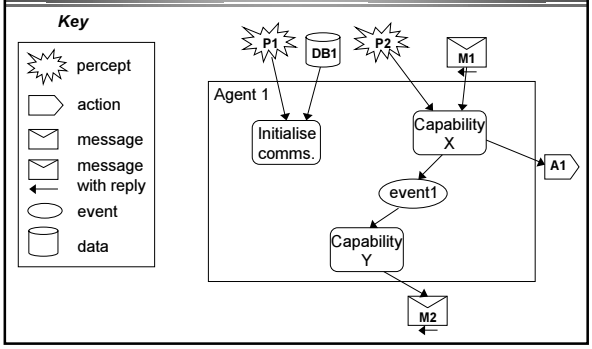
Capability descriptors

- Use functionalities as an initial list of capabilities
- May have some new capability descriptors to add
  - low level “library” capabilities
  - new included capabilities
- Using functionality descriptors as a base
  - make some fields more precise
  - add “included capabilities”
  - identify messages / percepts / events handled
  - identify messages sent and events posted
  - identify actions
  - identify interactions with other capabilities

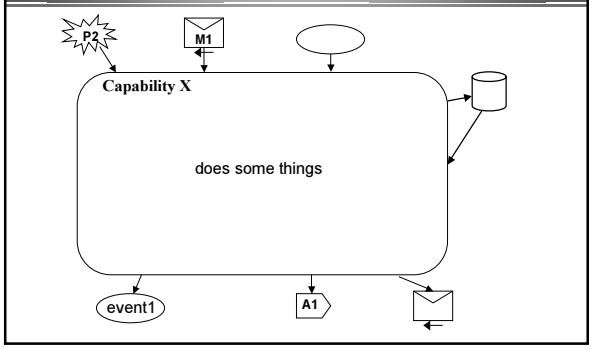
Agent Overview



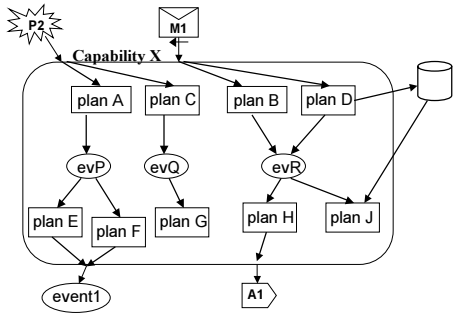
Zooming into the Capability



Capability Overview



Capability Overview



Coverage

- For this event (message / percept) will there always be an appropriate response - i.e. some plan defined which has a context condition which matches the situation?
- Or may there be some situations where there is no plan with a matching context condition - i.e. there will be no processing of the event?

Overlap

- Are there situations where there may be multiple responses to this event - i.e. the situation can be matched by more than one context condition?
- If so what is the prioritisation between the different plan types? How will that prioritisation work?

Event description template

- Name
- Description
- Purpose(s)
- Arguments (all data carried)
- Coverage
- Overlap

The percept and message descriptors should be augmented with arguments, coverage and overlap fields.

Defining the plans

- When should the plan be used?
  - what event is it triggered by?
  - what goal does it achieve?
  - in what situation is it applicable?
- What does the plan do?
- Under what conditions should it be aborted?
- What should be done if it fails?
  - two kinds of "failure": plan failure, and failure to achieve intended outcome (goal failure). Latter must be explicitly tested for.

Defining plan applicability

- What goal does the plan achieve? Or what event is it triggered by?
- For any goal / event there may be different plans for different situations
- Different situations defined according to context - checked via agent beliefs.
- Consider coverage and overlap

Defining the plan body

- Plan body consists of
  - subgoals or subtasks
  - internal plan methods
  - other code
- use subgoals wherever possible
- think whether you need to check success / failure via percepts?

Plan description template

- Name
- Description
- Event (including percepts and messages) handled
- Appropriate situation - precise context description
- Overlaps with - other plans which could be applicable at the same time.
- Procedure description - detailed pseudo code
- Data read and written

Data structure descriptions

Beliefs

- knowledge base beliefs
- program data structures
- static knowledge
- knowledge which is updated at runtime

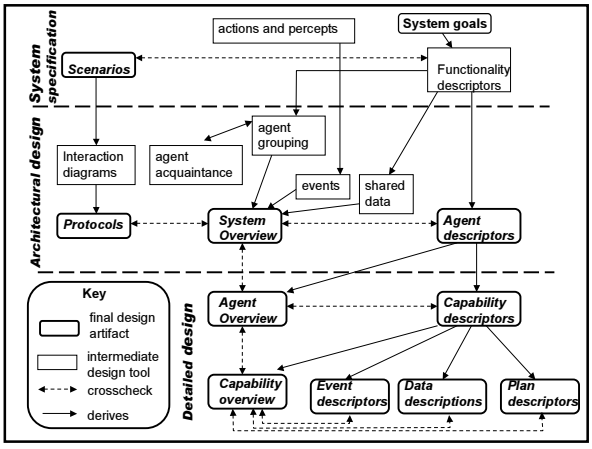
Other data

- objects in the system
- …

Design includes data - not particularly agent-oriented

Data Dictionary

- Keep track of all defined entities
  - agents
  - functionalities
  - plans
  - capabilities
  - events, messages, percepts
  - actions
- Organise alphabetically within types
- Choose names well



Exercise 1

Pretend you are building an agent system for meeting scheduling within an organisation. The system could potentially arrange meetings, book rooms, send reminder emails to participants about meetings, cancel meetings when a (key) participant is unable to attend, manage people's personal calendars, provide information about all meetings schedued, etc. The exact nature of what you define now is up to you. (Of course in a real situation, this would be established by discussing with the client, users, etc.). Ensure that you do define a system where there is some dynamic aspect to the environment, but try to keep it fairly limited so its easier to work with.

1. Take some central task in this application and sketch out some different plans/approaches for different situations.
2. **Identify a few system goals for some aspect of this application.**
3. **Identify some functionalities related to the goals you have identified.**
4. Identify data that is likely to be needed by these functionalities.
5. **Identify some percepts and actions.**
6. Identify which percepts are likely to be relevant to the functionalities you have identified
7. Which actions are the functionalities you identified likely to perform.
8. Fill out a functionality schema for one functionality.
9. **Write one or more scenarios involving the functionalities you have identified.**
10. Identify any extra functionalities required by your scenario(s).
11. Identify any additional data required by your scenario(s).
12. Indicate some plausible variations on the scenario for one of these scenarios.

Exercise 2

Inevitably these exercises cannot be done thoroughly as there is too little time, and additionally they rely on a previous phase for which there was also too little time. Nevertheless, they should help you to gain a more concrete understanding of the methodology than you otherwise would.

1. Indicate a set of functionalities (names and very brief descriptions) which you consider sufficient for a simple version of this application.
2. Write down the data stores/structures you expect to have in the system.
3. Do a data coupling diagram (as best you can from from your intuition as detailed information is not available)
4. **Determine appropriate agent types based on some grouping of functionalities.**
5. Identify the relevant percepts for one of your agents. What processing will be needed?
6. How many agents of each type will there be in the system?
7. **Do interaction diagrams for the scenario(s) you developed.**
8. Based on the previous question, specify the associated interaction protocol.(s)
9. Write the agent descriptor for one of your agent types..
10. **Do a system overview diagram for this (sub) system**

Exercise 3

1. Do an agent overview diagram for one of the agents in you system, showing the capabilities and interactions between them.
2. **Design some plans that enable these capabilities to provide the required functionality - summarize these in capability overview diagrams.**
3. **Provide plan descriptors for some of the plans (choose the most interesting ones).**
4. What event will trigger each of these plans? What information will the event carry?
5. **What "beliefs" or information is required by these plans? How/where will this information be kept? Indicate where it will be updated appropriately for use by your plans.**
6. Do a hand trace showing where the plans are used in particular scenarios.
7. Discuss where you could potentially add additional functionality to the system by adding further plans or capabilities. Discuss what would need to be added or changed to add this functionality.

Prometheus: A Methodology for Developing Intelligent Agents

Lin Padgham and Michael Winikoff

RMIT University, GPO Box 2476V, Melbourne, AUSTRALIA
Phone: +61 3 9925 2348
{linpa,winikoff}@cs.rmit.edu.au
http://www.cs.rmit.edu.au

Abstract. As agents gain acceptance as a technology there is a growing need for practical methods for developing agent applications. This paper presents the *Prometheus* methodology, which has been developed over several years in collaboration with Agent Oriented Software. The methodology has been taught at industry workshops and university courses. It has proven effective in assisting developers to design, document, and build agent systems. Prometheus differs from existing methodologies in that it is a detailed and complete (start to end) methodology for developing intelligent agents which has evolved out of industrial and pedagogical experience. This paper describes the process and the products of the methodology illustrated by a running example.

1 Introduction

As agents are gaining acceptance as a technology and are being used, there is a growing need for practical methods for developing agent applications. This paper presents the *Prometheus*[1] methodology for developing intelligent agent systems.

The methodology has been developed over the last several years in collaboration with Agent Oriented Software[2] (AOS). Our goal in developing Prometheus was to have a process with associated deliverables which can be taught to industry practitioners and undergraduate students who do not have a background in agents and which they can use to develop intelligent agent systems. To this end Prometheus is *detailed* and *complete* in the sense of covering all activities required in developing intelligent agent systems.

Our claim is that Prometheus is developed in sufficient detail to be used by a non-expert. Prometheus has been taught to an undergraduate class of (third year) students who successfully designed and implemented agent systems using JACK. A second year student over the summer vacation was given a description of the methodology and a description of an agent application (in the area of Holonic Manufacturing). With only (intentionally) limited support, the student was able to design and implement an agent system to perform Holonic Manufacturing using a simulator of a manufacturing cell.

[1] Prometheus was the wisest Titan. His name means "forethought" and he was able to foretell the future. Prometheus is known as the protector and benefactor of man. He gave mankind a number of gifts including fire. (*http://www.greekmythology.com/*)

[2] *http://www.agent-software.com*

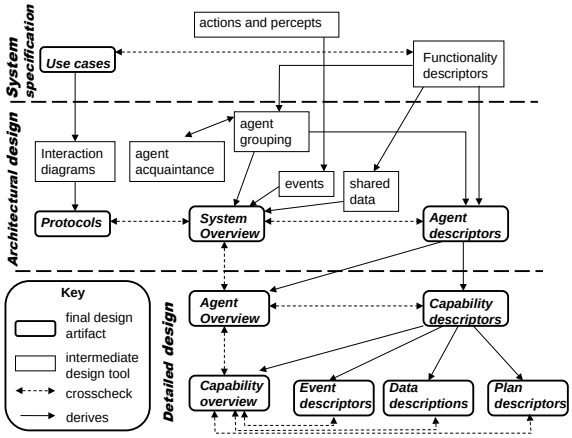
Unfortunately space limitations preclude a detailed comparison with the many existing methodologies. We simply note that Prometheus differs from existing methodologies [1–5, 7–18, 22, 23, 25] in that it:

- Supports the development of *intelligent* agents which use goals, beliefs, plans, and events. By contrast, many other methodologies treat agents as "simple software processes that interact with each other to meet an overall system goal" [6].
- Provides "start-to-end" support (from specification to detailed design and implementation) and a *detailed process*, along with design artifacts constructed and steps for deriving artifacts.
- Evolved out of practical industrial and pedagogical experience, and has been used by both industrial practitioners and by undergraduate students. By contrast, many other methodologies have been used only by their creators and often only on small (and unimplemented) examples.
- Provides hierarchical structuring mechanisms which allow design to be performed at multiple levels of abstraction. Such mechanisms are crucial to the practicality of the methodology on large designs.
- Uses an iterative process over software engineering phases rather than a linear "waterfall" model. Although the phases are described in a sequential fashion in this paper, the intention is *not* to perform them purely in sequence.
- Provides (automatable) cross checking of design artifacts.

Of the properties above, perhaps the most contentious is the first: many existing methodologies intentionally do not support intelligent agents, rather, they aim for generality and treat agents as black boxes. We believe that in this case, generality needs to be sacrificed in favour of usefulness. By specifically supporting the development of BDI-like agents we are able to provide detailed processes and deliverables which are useful to developers. Of course, this makes Prometheus less useful to those developing non-BDI-like agents. However, note that the initial stages of the methodology *are* appropriate for the design of any kind of multi agent system.

Although none of these properties is unique in isolation, their combination is, to the best of our knowledge, unique. We believe that these properties are all essential for a practical methodology that is usable by non-experts and accordingly the design of Prometheus was guided by these properties. Although Prometheus' contribution is the combination of these properties, this combination was achieved through careful design of the methodology. It is not possible to easily construct a methodology which has the above properties by combining methodologies that have some of them. For example, given a methodology that provides automated support but does not support intelligent agents and another methodology that supports intelligent agents but not provide automated cross-checking; it is not at all obvious how a hybrid methodology could be created that supports both features.

The *Prometheus* methodology consists of three phases. The *system specification phase* focuses on identifying the basic functionalities of the system, along with inputs (percepts), outputs (actions) and any important shared data sources. The *architectural design phase* uses the outputs from the previous phase to determine which agents the system will contain and how they will interact. The *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system.



The rest of this paper describes the methodology using a running example: an online bookstore that assists customers in finding and choosing books, manages deliveries, stock, customer and supplier information, and does selective advertising based on interests. Space limitations prevent us from describing in full detail all aspects of the Prometheus methodology. However, we hope that an understanding of its structure and some sense of its phases, deliverables, activities, and depth of detail can be gained.

2 System specification

Agent systems are typically situated in a changing and dynamic environment, which can be affected, though not totally controlled by the agent system. One of the earliest questions which must be answered is how the agent system is going to interact with this environment. In line with [21] we call incoming information from the environment "percepts", and the mechanisms for affecting the environment "actions".

As discussed in [24] it is important to distinguish between percepts and events: an event is a significant occurrence for the agent system, whereas a percept is raw data available to the agent system. Often percepts can require processing in order to identify events that an agent can react to. For example, if a soccer playing robot's camera shows a ball where it is expected to be then this percept is not significant. However, if the ball is seen where it is not expected then this *is* significant.

Actions may also be complex, requiring significant design and development outside the realm of the reasoning system. This is especially true when manipulation of physical effectors is involved. We shall not address percept processing and actions any further in this paper. Both can be done within the agent system (either in specific agents, or distributed) or outside of the agent part of the system. If done within the agent part of the system then the Prometheus methodology can be applied, otherwise existing methodologies can be used.

The online bookstore has the percepts of customers visiting the website, selecting items, placing orders (using forms), and receiving email from customers, delivery services and book suppliers. Actions are bank transactions, sending email, and placing delivery orders.

In parallel with discovering or specifying (which of these will depend on the situation) the percepts and actions the developer must start to describe what it is the agent system should do in a broader sense - the functionalities[3] of the system. For example, in order to define the book store we may need to define functionalities such as *"the book store will provide a personalised interface to customers"* and *"the book store will maintain its stock"*. These functionalities start to give an understanding of the system - some sense of its purpose.

It is important in defining functionalities that they be kept as narrow as possible, dealing with a single aspect or goal of the system. If functionalities are too broad they are likely to be less adequately specified leading to potential misunderstanding.

In defining a functionality it is important to also define the information that is required, and the information produced by it. The functionality descriptor produced contains a **name**, a short natural language **description**, a list of **actions**, a list of relevant **percepts**, **data used** and **produced** and a brief description of **interactions** with other functionalities. For example, the following describes the *welcomer* functionality in the online bookstore.

Welcomer: provides a customised response to the user when they log into the site.
Actions: provide link to status of existing orders, welcome by name, welcome as new user, query enjoyment of recent purchases, indicate special offers relevant to interests.
Percepts: Customer accesses site.
Data access: Reads customer information, special offers, and customer interactions data. Writes customer interactions data.
Interactions: No interactions with other functionalities.

While functionalities focus on particular aspects of the system, *use case scenarios* give a more holistic view of the system. The basic idea is borrowed from object oriented design. However, the use case scenarios are given slightly more structure.

The central part of a use case scenario in Prometheus is the sequence of steps describing an example of the system in operation. Each step is annotated with the name of the functionality responsible, as well as information used or produced. These annotations allow cross checking for consistency with the functionality descriptors.

The use case templates which we use contain an **identification number**, a brief natural language **overview**, an optional field called **context** which indicates when this scenario would happen, or the start point of the scenario, the **scenario** itself which is a sequence of steps, a summary of all the **information** used in the various steps, and a list of small **variations**. Because a scenario captures only one particular sequence of steps it can be useful to indicate small variations with a brief description. Any major variations should be a separate use case scenario.

3 Architectural design

The major decision to be made during the architectural design is which agents should exist. We assign functionalities to agents by analysing the artifacts of the previous phase

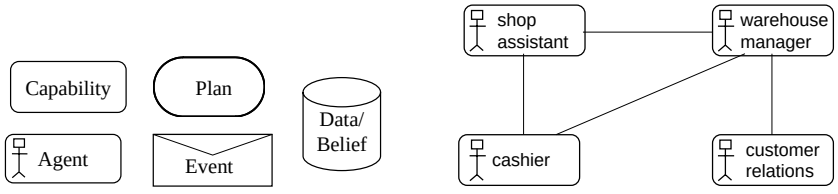
[3] A number of methodologies call these "roles". We prefer to avoid overloading the term since it has a similar, but non-identical, meaning in the context of teams of agents.

to suggest possible assignments of functionalities to agents. These are then evaluated according to the traditional software engineering criteria of coherence and coupling.

The process of identifying agents by grouping functionalities involves analysing the reasons for and against groupings of particular functionalities. If functionalities use the same data it is an indication for grouping them, as is significant interaction between them. Reasons against groupings may be clearly unrelated functionality or existence on different hardware platforms. More generally, we seek to have agents which have strong coherence and loose coupling.

It can be useful at this stage to draw a matrix having all functionalities on one axis and the properties or relationships on the other axis. Specific properties and relationships that are useful in deriving groupings of functionalities are whether two functionalities are *related*, whether they are *clearly unrelated*, the *data used*[4] and *data produced* as well as *information received* from other functionalities and data that is *written by two (or more) functionalities*. The last two columns can be derived from the information in the previous columns.

In order to evaluate a potential grouping for coupling we use an agent acquaintance diagram. This diagram simply links each agent with each other agent with which it interacts. A design with fewer linkages is less highly coupled and therefore preferable. The design for the book store depicted below (on the right) is reasonable, since it indicates low coupling. A design which produced an acquaintance diagram where each agent was linked to every other agent would be highly undesirable. Note that Prometheus uses a consistent notation to depict agents, events, plans, capabilities, etc. This notation is summarised below, on the left.



A simple heuristic for assessing coherence is whether an agent has a simple descriptive name which encompasses all the functionalities without any conjunctions (“and”). For example, the shop assistant agent combines the functionalities of visit manager, client welcomer, query processor, pro-active helper, and customer DB manager; yet it has a simple descriptive name.

Once a decision has been made as to which agents the system should contain it is possible to start working out and describing some of the necessary information about agents. The high level information about each agent is contained in the form of an agent descriptor, similar to functionality descriptors. Questions which need to be resolved about agents at this stage include: How many agents of this type will there be (singleton, a set number, or multiple based on dynamics of the system, e.g. one sales assistant agent per customer)? What is the lifetime of the agent? If they are created or destroyed during system operation (other than at start-up and shut-down), what triggers this? Agent initialisation - what needs to be done? Agent demise - what needs to be

[4] Both *read* (in the case of data stores) and *received* (in the case of events and messages).

done? What data does this agent need to keep track of? What events will this agent react to?

Agent descriptors contain the above information plus the **name** and **description** of the agent, a brief description of ways in which it **interacts** with other agents and a list of the **functionalities** which are incorporated within this agent. For example consider the following agent descriptor:
Name: Shop assistant agent
Description: greets customer, follows through site, assists with finding books
Cardinality: 1/customer. Instantiated on customer arrival at site
Functionalities included: visit manager, client welcomer, query processor, pro-active helper, customer DB manager.
Reads data: user profile, client orders
Writes data: user profile
Interacts with: cashier (to pay for purchase); warehouse manager (to get price, availability and to hand over order for shipping)

Consistency checking should be done to ensure that agent descriptors are consistent with the set of functionality descriptors which have been incorporated within the agent. Particular items to check are the information, and the agent interactions. If a functionality is listed as interacting with another functionality, then this should translate into an agent interaction, unless the two functionalities are incorporated within the same agent.

At this stage of the design it is important to identify what **events** (i.e. significant occurrences) will be generated as a result of information from the environment (the percepts), either directly or after processing. These are the things the agents will notice, which will cause them to react in some way. A decision must be made as to which agents will react to which events.

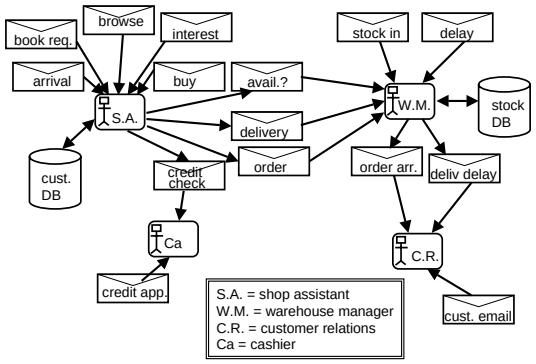
In order to accomplish the various aims of the system agents will also send messages to each other. These must also be identified at this stage. It is also necessary to identify what information fields will be carried in these messages, as this forms the interface definition between the agents.

If the implementation platform does not provide specialised message types, these must be specified precisely at this stage to enable modularity in the development of the detailed design for the individual agents.

Shared data objects (if any) must also be identified at this stage. A good design will minimise these, but there may be situations where it is reasonable to have shared data objects. If multiple agents will be writing to shared data objects this will require significant additional care for synchronisation (as agents operate concurrently with each other). Often what at first appears to be a shared data object can be reconceptualised as a data source managed by a single agent, with information provided to other agents as they need it. Alternatively each agent may have its own version of the information, without there being any need for a single centralised data object. Data objects should be specified using traditional object oriented techniques.

The **system overview diagram** ties together the agents, events and shared data objects. It is arguably the single most important artifact of the entire design process, although of course it cannot really be understood fully in isolation. A system overview diagram for the running example is below. By viewing this diagram we obtain a general

understanding of how the system as a whole will function. Messages between agents can include a reply, although this is not shown explicitly on the diagram. Looking further at agent descriptors provides any additional detail needed to understand the high level functioning of the system.



The final aspect of the architectural design is to specify fully the **interaction** between agents. Interaction diagrams are used as an initial tool for doing this, while fully specified interaction protocols are the final design artifact.

Interaction diagrams are borrowed directly from object oriented design, showing interaction between agents rather than objects. One of the main processes for developing interaction diagrams is to take the use cases developed in the specification phase and to build corresponding interaction diagrams. Wherever there is a step in the use case which involves a functionality from a new agent there must be some interaction from a previously involved agent to the newly participating agent. While it is not possible to automatically derive the interaction diagrams from the use cases, substantial consistency checking is possible. Figure 1 (left) shows an interaction diagram for a scenario of buying a book at our electronic bookstore. It depicts the user requesting a particular book from the sales assistant which checks the details with the warehouse then replies. The user decides to buy the book and places an order; the sales assistant checks for delivery options, confirms them with the user, checks the user's credit card details with the cashier, and then places the order and thanks the user. In addition to deriving interaction diagrams from use cases, each of the major environmental events should have an associated interaction diagram.

Interaction diagrams, like use cases, give only a partial picture of the system's behaviour. In order to have a precisely defined system we progress from interaction diagrams to **interaction protocols** which define precisely which interaction sequences are valid within the system.

Figure 1 (right) shows the protocol for the credit check portion of the interaction diagram shown in figure 1 (left). Because protocols must show all variations they are often larger than the corresponding interaction diagram and may need to be split into smaller chunks. We use the AUML notation [18] to specify protocols,

Consistency checking should be done between protocols and interaction diagrams, the system overview diagram, and use cases. With the appropriate tools, much of this consistency checking can be automated.

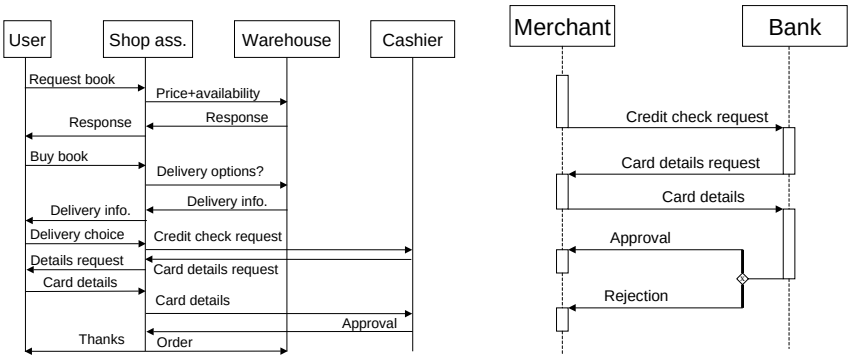


Fig. 1. Example Interaction Diagram (left) and Protocol (right)

4 Detailed design

Detailed design focuses on developing the internal structure of each of the agents and how it will achieve its tasks within the system. It is at this stage of the design that the methodology becomes specific to agents that use user-defined plans, triggered by goals or events, such as the various implementations of Belief, Desire, Intention (BDI) systems (e.g. PRS, dMARS, JAM, or JACK). A number of details regarding the implementation platform also become evident at this stage when looking at any particular design. However, the principles are easily adapted to the specifics of whichever development platform has been chosen, as long as it is within the broad general category of agents which use plans and react to events.

The focus of the detailed design phase is on defining capabilities (modules within the agent), internal events, plans and detailed data structures. A progressive refinement process is used which begins by describing agents' internals in terms of capabilities. The internal structure of each capability is then described, optionally using or introducing further capabilities. These are refined in turn until all capabilities have been defined. At the bottom level capabilities are defined in terms of plans, events, and data.

The functionalities from the specification phase provide a good initial set of **capabilities**, which can be further refined if desired. Sometimes there is also functionality akin to "library routines" which is required in multiple places - either within multiple agents, or within multiple capabilities within a single agent. Such functionality should also be extracted into a capability which can then be included into other capabilities or agents as required.

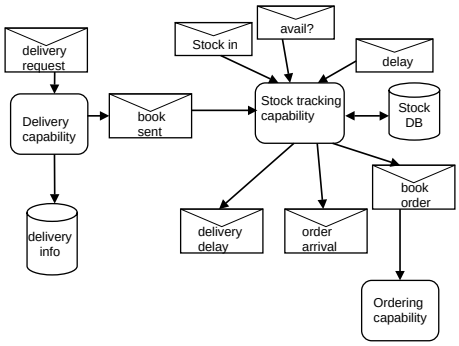
Capabilities are allowed to be nested within other capabilities and thus this model allows for arbitrarily many layers within the detailed design, in order to achieve an understandable complexity at each level.

Each capability should be described by a capability descriptor which contains information about the external interface to the capability - which **events** are **inputs** and which events are **produced** by (as inputs to other capabilities). It also contains a natural language **description** of the functionality, a unique descriptive **name**, information

regarding **interactions** with other capabilities, or **inclusions** of other capabilities, and a reference to **data** read and written by the capability. We use structured capability descriptor forms containing the above fields.

The **agent overview diagram** provides the top level view of the agent internals. It is very similar in style to the system overview diagram, but instead of agents within a system, it shows capabilities within an agent. This diagram shows the top level capabilities of the agent and the event or task flow between these capabilities, as well as data internal to the agent. By reading the relevant capability descriptors, together with the diagram, it is possible to obtain a clear high level view of how the modules within the agent will interact to achieve the overall tasks of the agent as described in the agent descriptor from the architectural design.

The agent overview diagram below is for a warehouse manager agent in the electronic bookstore. This agent has the capabilities of tracking stock, placing orders for new stock and organising delivery of books to clients.



A further level of detail is provided by capability diagrams which take a single capability and describe its internals. At the bottom level these will contain plans, with events providing the connections between plans, just as they do between capabilities and between agents. At intermediate levels they may contain nested capabilities or a mixture of capabilities and plans. These diagrams are similar in style to the system overview and agent overview diagram, although plans are constrained to have a single incoming (triggering) event.

Just as the agent overview diagram should be checked for consistency with the system overview (in terms of incoming and outgoing events), so each capability overview diagram should be checked against its enclosing context - either the agent overview, or another capability overview.

The final design artifacts required are the individual plan, event and data descriptors. These descriptions provide the details necessary to move into implementation. Exactly what are the appropriate details for these descriptors will depend on aspects of the implementation platform. For example if the context in which a plan type is to be used is split into two separate checks within the system being used (as is the case in JACK) then it is appropriate to specify these separately in the descriptor. Fields regarding what information an event carries assumes that events are composite objects able to carry information, and so on.

The plan descriptors we use provide an **identifier**, the **triggering event type**, the **plan steps** as well as a short natural language **description**, a **context** specification indicating when this plan should be used and a list of **data** read and written.

Event descriptors are used to fully specify all events, including those identified earlier. The event descriptor should identify the **purpose** of the event and any **data** that the event carries. We also indicate for each event whether it is expected to be *covered* and whether it is expected to be *unambiguous*. An event is *covered* if there is always at least one handling plan which is applicable; that is, for any situation, at least one of the matching plans will have a true context condition. An event is *unambiguous* if there is always at *most* one handling plan which is applicable.

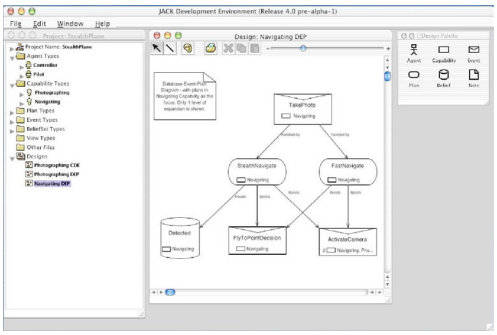
Data descriptors should specify the fields and methods of any classes used for data storage within the system. If specialised data structures are provided for maintaining beliefs, these should also be specified.

An additional artifact that is completed (and checked) at this point is the **data dictionary**. The data dictionary should be started at the beginning of the project and developed further at each stage. The data dictionary is important in ensuring consistent use of names (for example, what is called “delivery info” in one place in the design should not be called “deliv. information” elsewhere). One option is to organise the data dictionary into separate sections for agents, capabilities, plans, events and data, organised alphabetically within sections. The other option is to have a flat alphabetical structure. With tool support multiple views (automatically generated) can be provided.

5 Discussion and conclusions

We have briefly described the key aspects of the Prometheus methodology. The methodology has been in use for several years and has been taught in industry workshops (most recently at the Australian AI conference, 2001). The methodology has been in use for several years as a teaching tool. The feedback we have received indicates that it provides substantial guidance for the process of developing the design and for communicating the design within a work group. With student projects it is abundantly clear that the existence of the methodology is an enormous help in thinking about and deciding on the design issues, as well as conveying the design decisions.

One of the advantages of this methodology is the number of places where automated tools can be used for consistency checking across the various artifacts of the design process. For example, the input and output events for an agent must be the same on the system overview diagram and on the agent overview diagram. Agent Oriented Software has constructed a support tool for the methodology



that allows design diagrams to be drawn and generates corresponding skeleton code (in JACK).

We are also investigating how some of the design artifacts, such as the protocol definitions, and the capability diagrams, can be used for providing debugging and tracing support within the implemented system [20]. Having a design methodology which can be used through to testing and debugging is clearly advantageous in terms of an integrated and complete methodology.

Other areas for future work include: clearer integration of goals as a first class concept (currently goals are implicit in functionalities), extensions to the graphical notation to allow percepts, actions, goals, and (some) sequencing information to be specified; introduction of social concepts (teams, roles) and a clarification of the distinction between functionalities and roles; and investigating the commonalities and differences with various extensions of UML to agents [18, 19]. Additionally, we intend to integrate Prometheus with the agent concepts we have identified [24].

Acknowledgements: We would like to acknowledge the support of Agent Oriented Software Pty. Ltd. and of the Australian Research Council (ARC) under grant CO0106934. We would also like to thank James Harland and Jamie Curmi for comments on drafts of this paper.

References

1. F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *Int Journal of Cooperative Information Systems*, 6(1):67–94, 1997.
2. B. Burmeister. Models and methodology for agent-oriented analysis and design. Working Notes of the KI'96 Workshop on AgentOriented Programming and Distributed Systems, 1996.
3. G. Bush, S. Cranefield, and M. Purvis. The Styx agent methodology. The Information Science Discussion Paper Series 2001/02, Department of Information Science, University of Otago, New Zealand., Jan. 2001. Available from *http://divcom.otago.ac.nz/infosci*.
4. G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massonet. Agent oriented analysis using MESSAGE/UML. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 101–108, 2001.
5. A. Collinot, A. Drogoul, and P. Benhamou. Agent oriented design of a soccer robot team. In *Proceedings of ICMAS'96*, 1996.
6. S. A. DeLoach. Analysis and design using MaSE and agentTool. In *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, 2001.
7. S. A. DeLoach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.
8. A. Drogoul and J. Zucker. Methodological issues for designing multi-agent systems with machine learning techniques: Capitalizing experiences from the robocup challenge. Technical Report LIP6 1998/041, Laboratoire d'Informatique de Paris 6, 1998.
9. M. Elammari and W. Lalonde. An agent-oriented methodology: High-level and intermediate models. In G. Wagner and E. Yu, editors, Proc. of the 1st Int. Workshop. on Agent-Oriented Information Systems., 1999.

10. N. Glaser. The CoMoMAS methodology and environment for multi-agent system development. In C. Zhang and D. Lukose, editors, *Multi-Agent Systems Methodologies and Applications*, pages 1–16. Springer LNAI 1286, Aug. 1996. Second Australian Workshop on Distributed Artificial Intelligence.
11. C. Iglesias, M. Garijo, and J. González. A survey of agent-oriented methodologies. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.
12. C. A. Iglesias, M. Garijo, J. C. González, and J. R. Velasco. Analysis and design of multiagent systems using MAS-commonKADS. In *Agent Theories, Architectures, and Languages*, pages 313–327, 1997.
13. E. A. Kendall, M. T. Malkoun, and C. H. Jiang. A methodology for developing agent based systems. In C. Zhang and D. Lukose, editors, *First Australian Workshop on Distributed Artificial Intelligence*, 1995.
14. D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. In *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96). LNAI 1193*. Springer-Verlag, 1996.
15. D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In R. van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996.
16. J. Lind. A development method for multiagent systems. In *Cybernetics and Systems: Proceedings of the 15th European Meeting on Cybernetics and Systems Research, Symposium "From Agent Theory to Agent Implementation"*, 2000.
17. J. Mylopoulos, J. Castro, and M. Kolp. Tropos: Toward agent-oriented information systems engineering. In *Second International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS2000)*, June 2000.
18. J. Odell, H. Parunak, and B. Bauer. Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence.*, 2000.
19. M. Papasimeon and C. Heinze. Extending the UML for designing JACK agents. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*, Aug. 2001.
20. D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. To appear in the proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems, 2002.
21. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
22. O. Shehory and A. Sturm. Evaluation of modeling techniques for agent-based systems. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 624–631. ACM Press, May 2001.
23. L. Z. Varga, N. R. Jennings, and D. Cockburn. Integrating intelligent systems into a cooperating community for electricity distribution management. *Int Journal of Expert Systems with Applications*, 7(4):563–579, 1994.
24. M. Winikoff, L. Padgham, and J. Harland. Simplifying the development of intelligent agents. In M. Stumptner, D. Corbett, and M. Brooks, editors, *AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence*, pages 555–568. Springer, LNAI 2256, Dec. 2001.
25. M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.

Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents

Lin Padgham Michael Winikoff
RMIT University
{linpa,winikoff}@cs.rmit.edu.au

Abstract

Agents are a powerful technology with many significant applications. A key issue in getting the technology into mainstream software development is the development of appropriate methodologies *for engineering agent-oriented software. This paper presents the* Prometheus *methodology, which has been developed over several years in collaboration with Agent Oriented Software. The methodology has been taught at industry workshops and university courses. It has proven effective in assisting developers to design, document, and build agent systems. Prometheus is a detailed and complete (start to end) methodology for developing intelligent agents which has evolved out of industrial and pedagogical experience, and which is supported by tools. This paper gives a brief overview of the design process as a whole and then discusses envisaged tool support as well as the implemented tool prototypes we have been using.*

1 Position Statement

Agents are a powerful technology with many significant applications, both demonstrated and potential [8, 7]. A key issue in getting the technology into the mainstream of software development is the development of appropriate *methodologies* for engineering agent-oriented software. This paper motivates and presents the *Prometheus*[1] methodology for developing intelligent agent systems.

We consider a methodology to encompass (i) a set of concepts used; (ii) notations for modelling aspects of the software (requirements, designs, implementation); and (iii) a process that is followed in order to produce the software.

The Prometheus methodology has been developed over the last several years in collaboration with Agent Oriented Software[2] (AOS). Our goal in developing Prometheus was to have a process with associated deliverables which can be taught to industry practitioners and undergraduate students who do not have a background in agents and which they can use to develop intelligent agent systems. To this end Prometheus is *detailed* and *complete* in the sense of covering all the stages of software development as applied to agent systems.

We believe that particular strengths of the Prometheus methodology include:

- Provision of "start-to-end" support (from specification to detailed design, implementation and testing), including a *detailed process*, along with design artifacts constructed and steps for deriving artifacts.
- Hierarchical structuring mechanisms which allow design to be performed at multiple levels of abstraction. Such mechanisms are crucial to the practicality of the methodology for large applications.
- The structured nature of the design artifacts facilitates development of tool support, especially automated cross checking of design artifacts, and automated provision of skeleton artifacts at certain stages. Partial tool support already exists.
- Support for detailed design of the internals of *intelligent* agents. This necessarily makes increased assumptions about the implementation platform. The detailed design assumes a plan based system where agents react to events, based on their beliefs about the situation. It is particularly well suited to BDI type systems.
- The fact that the methodology has evolved out of practical industrial and pedagogical experience, and thus addresses issues in designing agent systems that have been experienced by both industrial practitioners and undergraduate students.

Prometheus also uses an iterative process over the phases described in this paper, rather than a linear "waterfall"

[1] Prometheus was the wisest Titan. His name means "forethought" and he was able to foretell the future. Prometheus is known as the protector and benefactor of man. He gave mankind a number of gifts including fire. (*http://www.greekmythology.com/*)

[2] *http://www.agent-software.com*

DRAFT

model, although the amount of work on the earlier phases decreases as the design progresses, just as the amount of work on later phases increases.

Designs for large systems are almost always developed incrementally with many revisions. When revising any artefact, be it documentation, code, or design, it is easy to introduce inconsistencies and minor errors. We have found the prototype tool we have developed extremely useful for checking and maintaining design consistency across varying levels of detail. The automatic generation of skeleton code from design artifacts in the tool prototype is also extremely useful, and has encouraged students to do design prior to coding.

We have worked with development of agent software for eight years and have during this time had a wealth of experience in trying to teach students to build such systems. The Prometheus methodology has partially grown out of this experience and we have noticed an enormous difference in the last few years, in the ability of our students to develop agent systems. Previously they would flounder and end up building a system which made little real use of agents. using Prometheus, third year undergraduates are now able to build reasonable agent systems in a one semester course. Over the last summer vacation a second year student was given a description of the methodology and a description of an agent application (in the area of Holonic Manufacturing). With only (intentionally) limited support, the student was able to design and implement an agent system to perform Holonic Manufacturing using a simulator of a manufacturing cell. Although this is only anecdotal evidence that the methodology is helpful, it has had a very significant effect on our ability to teach students to program agent systems.

The methodology is also taught to industry software developers who are starting to use the JACK intelligent agents development environment [2], and has been successful in introducing them to methods to assist them in design of agent applications.

In this paper we give a brief overview of the design process as a whole, focussing in somewhat more detail on the *Architectural Design*, in order to give some flavour of the methodology. We discuss envisaged tool support as well as the implemented tool prototypes we have been using. There is not room in a paper of this length for much comparison with other methodologies, but we mention this briefly in section 4, in particular discussing the relationship to UML.

2 The Prometheus Methodology

The *Prometheus* methodology [15] includes three design phases, where artifacts are produced which are used in both production of skeleton code for implementation, and for debugging and testing. The *system specification phase* focuses on identifying the basic functionalities of the system, along

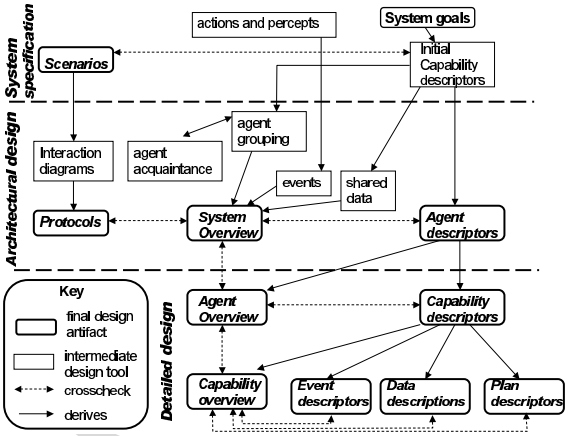


Figure 1. Phases, artifacts and relationships in the design process

with inputs (percepts), outputs (actions) and any important shared data sources. The *architectural design phase* uses the outputs from the previous phase to determine which agents the system will contain and how they will interact. The *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system. Figure 2 indicates the main design artifacts arising from each of these phases as well as some of the intermediary items and relationships between items.

Tools from OO analysis and design are adapted in several places as appropriate. For example, Prometheus' scenarios are a variant of the scenario part of UML's use cases. while our interaction diagrams are essentially UML sequence diagrams.

We describe the methodology using a running example: an online bookstore that assists customers in finding and choosing books, manages deliveries, stock, customer and supplier information, and does selective advertising based on interests. Space limitations prevent us from describing in detail all aspects of the Prometheus methodology. However, we hope that some understanding of its structure, phases, artifacts and activities can be gained. We are not able to cover testing and debugging in this paper, but some of this is covered in [17].

2.1 System specification

Before starting to design or build any software system there is discussion with clients, managers or other stakeholders regarding the general purpose of the system. We do not cover the preliminary part of this discussion, but start from the point at which it is possible to start specifying some of the details arising out of such discussions.

DRAFT

Agent systems are typically situated in a changing and dynamic environment, which can be affected, though not totally controlled by the agent system. One of the earliest questions which must be answered is how the agent system is going to interact with this environment. We call incoming information from the environment "percepts", and the mechanisms for affecting the environment "actions".

As discussed in [21] it is important to distinguish between percepts and events: an event is a significant occurrence for the agent system, whereas a percept is raw data available to the agent system. Often percepts can require processing in order to identify events that an agent can react to. For example, if a soccer playing robot's camera shows a ball where it is expected to be then this percept is not significant. However, if the ball is seen where it is not expected then this *is* significant.

Actions may also be complex, requiring significant design and development outside the realm of the reasoning system. This is especially true when manipulation of physical effectors is involved. We shall not address percept processing and actions any further in this paper, although identification and specification of these are an important part of the initial and ongoing design process.

The online bookstore's percepts and events include customers visiting the website, selecting items, placing orders using forms, and receiving email from customers, delivery services and book suppliers. Actions include bank transactions, sending email, and placing delivery orders.

In parallel with identifying or specifying (which of these will depend on the situation) the percepts and actions the developer must start to describe what it is the agent system should do in a broader sense - the goals and functionalities[3] of the system. For example, in order to define the book store we may have goals such as *maintain stock levels* and *provide competitive prices* with related functionalities such as *"stock ordering"*, *"stock monitoring"*, *"competitor monitoring"* and *"pricing review"*. These goals and functionalities start to give an understanding of the system.

It is important in defining functionalities that they be kept as narrow as possible, dealing with a single aspect or sub-goal of the system. If functionalities are too broad they are likely to be less adequately specified leading to potential misunderstanding.

In defining a functionality it is important to also define information required and produced by it. The functionality descriptor contains a name, a short natural language description, a list of actions, a list of relevant percepts, data used and produced and a brief description of interactions with other functionalities.[4] For example, the following describes the *welcoming* functionality in the online bookstore.

NAME: Welcoming
Description: Welcomes a new visitor to the world wide web site (with personalised information if possible).
Percepts/events/messages: CustomerArrived (message), CustomerInformation (message)
Messages sent: CustomerInformationRequest (message), CustomisedWWWPage (message),
Actions: DisplayCustomisedWWWPage
Data used: CustomerDB, CustomerOrders
Interactions: CustomerManager (via CustomerInformationRequest, CustomerInformation) OnlineInteraction (via CustomisedWWWPage, CustomerArrived)

Each functionality should be linked to some system goal, while each goal should result in one or more functionality, although there may not be a one to one mapping.

While functionalities focus on particular aspects of the system, *scenarios* give a more holistic view of system processing. The basic idea is borrowed from object oriented design. However, the scenarios are given slightly more structure.

The central part of a scenario in Prometheus is the sequence of steps describing an example of the system in operation. Each step in the scenario is one of the following:

- incoming event/percept (→ receiving functionality)
- message (sender → receiver)
- activity[5] (functionality)
- action (functionality)

These steps can optionally have data read and data written noted.

Our scenario templates contain an identification number, a brief natural language overview, an optional field called context which indicates the situations in which this scenario could occur, the sequence of steps, a summary of all the information used in the various steps, and a list of small variations. Because a scenario captures only one particular sequence of steps it can be useful to indicate small variations with a brief description. Any major variations should be a separate scenario.

Scenario: Book Order
Overview: The user orders a book. Delivery options are explored and then confirmed (with an OrderRequest). The books are shipped, stock updated, and the user notified.
Context: Assumes the book is in stock.
Steps:

1. EVENT BookOrder (→ Online Interaction)

[3] A number of methodologies call these "roles". We prefer to avoid overloading the term since it has a similar, but non-identical, meaning in the context of teams of agents.

[4] Any of these except name and description may be empty.

[5] We use the term *activity* to denote anything done within the functionality - typically some kind of computation

DRAFT

2. DeliveryOptionQuery (Online Interaction → Transport Information)
3. DeliveryOptions (Transport Information → Online Interaction) Data read: Transport DB
4. Obtain preferred delivery option (Online Interaction)
5. MakePayment (Online Interaction → Sales Transaction)
6. ACTION BankTransaction (Sales Transaction)
7. PlaceOrder (Sales Transaction → Order Handling)
8. Register order (Order Handling) Writes data: CustomerOrders
9. ACTION EmailCourierCompany (Order Handling)
10. DecreaseStock (Order Handling → Stock Manager)
Variations: steps 9 (email courier) and 10 (decrease stock) replaced with notification of delay (Order Handling to Customer Contact) and then placing an order for more stock (Order Handling to Stock Manager).

2.2 Architectural design

A major decision to be made during architectural design is which agent types the system should have. We assign functionalities to agents by analysing the artifacts of the previous phase to suggest possible assignments of functionalities to agents. These are then evaluated according to the traditional software engineering criteria of cohesion and coupling.

Once we have decided upon the agents in the system we identify which agents react to which percepts or environmental events, as well as which agents perform particular actions on the external environment. In addition we specify the messages that are sent between agents, and determine the major data repositories. These items form the overall design of the system and are depicted in the *system overview diagram*. The system overview diagram is perhaps the single most important product of the design process. It ties together agents, data, external input and output, and shows the communication between agents.

The system overview diagram shows the pathways of communication - which agents talk to which other agents - but not the *timing* of communication – which messages are followed by which other messages. An indication of the timing of communication is captured initially in the scenarios which describe processing. This can then be described in the form of agent interaction diagrams which show message passing between agents. Like scenarios, these depict one possible sequence of messages between agents. In order to describe all possible interactions we develop interaction protocols, depicted using the Agent UML (AUML) notation [14].

The process of identifying agents by grouping functionalities involves analysing the reasons for and against groupings of particular functionalities. If functionalities use the same data it is an indication for grouping them, as is significant interaction between them. Reasons against groupings may be clearly unrelated functionality or existence on different hardware platforms. More generally, we seek to have agents which have strong internal cohesion and loose coupling to other agents.

The main reasons for combining functionalities into a single agent are:

- The functionalities seem related - it "makes sense" to group them. For example, Competitor Monitoring and Pricing Review functionalities in the electronic bookstore example.
- The functionalities require a lot of the same information. If grouped into a single agent this can then be represented in internal agent data structures. If separate agents are used the information must be passed via messages unless a shared data store is used which is not usually a good design decision.

Reasons for *not* grouping functionalities include:

- The functionalities are clearly unrelated
- The functionalities exist on different hardware platforms
- Security and privacy - if data associated with one functionality should not be available to another functionality.
- Modifiability - if a functionality will change, or will be modified by different people.

One technique that we use to systematically examine the properties which lead to coupling is the *data coupling diagram*. A data coupling diagram consists of the functionalities and all identified data (not only persistent data, but also data the functionalities require to fulfil their job). Directed links are then inserted between functionalities and data, where an arrow pointing towards the data indicates the data is *produced or written by* that functionality, whereas an arrow pointing towards the functionality indicates the data is *used by* the functionality. A double-headed arrow indicates that the functionality both uses and produces the data. An example data coupling diagram[6] for the bookstore showing one possible grouping of functionalities into agents is shown in figure 2.

The diagram can be checked to ensure that all data is produced somewhere, unless it has been determined that it is provided externally on system start-up and is static. Also, the existence of data that is produced but not used usually indicates an omission elsewhere in the design.

[6]This particular diagram is actually reached after some analysis and refinement from the diagram first produced.

DRAFT

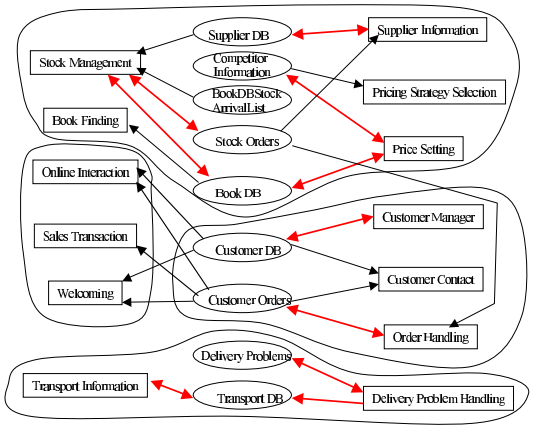


Figure 2. Example Data Coupling Diagram for the Book Store

The diagram can be assessed visually for groupings which are linked by their data use. Each such grouping must also be assessed for cohesion and with respect to whether there is some reason to keep the functionalities separated. It is one mechanism to assist in analysis of how functionalities may be combined.

To evaluate a potential grouping for coupling we use an agent acquaintance diagram. This diagram simply links each agent with each other agent with which it interacts. A design with fewer linkages is less highly coupled and therefore preferable.

A simple heuristic for assessing cohesion is whether an agent has a simple descriptive name which encompasses all the functionalities without any conjunctions ("and"). For example, the SalesAssistant agent combines the functionalities of OnlineInteraction, SalesTransaction and Welcoming; yet it has a simple descriptive name. This rule of thumb is obviously not universally applicable, however, we have found it to be a useful (and quick) way of assessing cohesion in proposed designs.

Once a decision has been made regarding agent types, high level information can be recorded in agent descriptors, similar to functionality descriptors. Questions which need to be resolved about agents at this stage include:

- How many agents of this type will there be (singleton, a set number, or multiple based on dynamics of the system, e.g. one sales assistant agent per customer)?
- What is the lifetime of the agent? If they are created or destroyed during system operation (other than at start-up and shut-down), what triggers this?
- Agent initialisation - what needs to be done?
- Agent demise - what needs to be done?
- What data does this agent need to keep track of?
- What events will this agent react to?

In addition we extract from the functionality descriptors, information about goals, percepts and events, actions, data and messages and add these to the agent descriptor.

For example consider the following agent descriptor from our electronic bookstore example:

Name: Sales Assistant agent
Description: greets customer, follows through site, assists with finding books
Cardinality: one/customer.
Lifetime: Instantiated on customer arrival at site. Demise when customer logs out or after inactivity period.
Initialisation: Obtains cookie. Reads Customer DB.
Demise: Closes open DB connections.
Functionalities included: Online Interaction, Sales Transaction, Welcomer, Book Finder.
Uses data: Customer DB, Customer Orders, Book DB.
Produces data: Customer preferences, orders, queries
Goals: Welcome customer; Update customer details; Respond to queries; Facilitate purchases;
Events responded to: new arrival; customer query; customer purchase; credit check response customer response;
Actions: Display information to customer (greetings, book info, info requests, Display customised WWW page, RequestCreditCheck messages
Interacts with: Warehouse Manager (book request protocol), Delivery Manager (order protocol, order query protocol), Customer Manager (customer information query protocol, customer information update protocol)

Having identified the agent types we proceed to capture the top level structure of the system using a system overview diagram. We identify *events* (i.e. significant occurrences) that will be generated as a result of information from the environment (the percepts), and will be noticed by the agents, and *shared data objects*.

A good design will minimise shared data objects, although there may be situations where this makes sense. If multiple agents will be writing to shared data objects this will require significant additional care for synchronisation (as agents operate concurrently with each other). Often shared data objects can be avoided by having the data source managed by a single agent, with information provided to other agents as needed. Sometimes each agent can have its own version of the information, without there being any need for a single centralised data object. Data objects could be specified using traditional object oriented techniques or database design techniques as appropriate.

The *system overview diagram* ties together the agents, events and shared data objects. It is arguably the single most

DRAFT

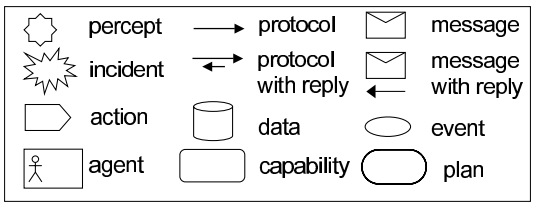


Figure 3. Notation for System Overview and other diagrams

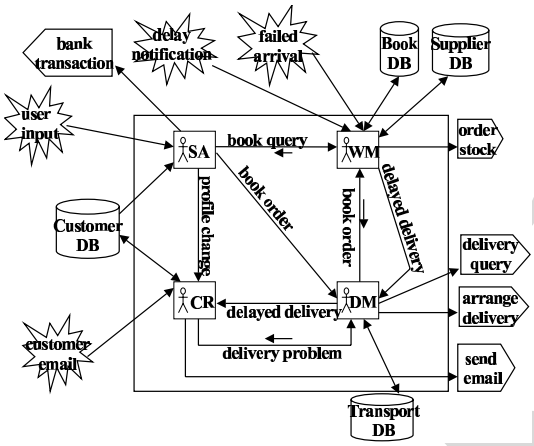


Figure 4. Partial system overview diagram for electronic bookstore

important artifact of the entire design process, although of course it cannot really be understood fully in isolation. By viewing a system overview diagram we obtain a general understanding of how the system as a whole will function, including interactions between agents. Agent descriptors and protocols provide additional detail needed to understand the high level functioning of the system. Similar diagrams are used at lower levels of the system as it is broken down in a hierarchical fashion. Figure 3 shows this notation while figure 4 provides a (partial) system overview diagram for the electronic bookstore.

The final aspect of the architectural design is to specify fully the *interaction* between agents. Interaction diagrams are used as an initial tool for doing this, while fully specified interaction protocols are the final design artifact.

Interaction diagrams are borrowed directly from object oriented design (UML's sequence diagrams), showing interaction between agents rather than objects. An advantage of the highly structured scenarios developed in the specification phase is that they can be used directly for producing interaction diagrams. Often at this stage there is some modification and refinement of scenarios to produce simpler interaction diagrams.

Interaction diagrams, like scenarios, give only a partial picture of the system's behaviour. In order to have a precisely defined system we progress from interaction diagrams to *interaction protocols*, specified in AUML, which define precisely which interaction sequences are valid within the system. These are generalisations of the interaction diagrams. At each point that a message is sent, the question must be asked as to whether there are other choices that could be made - no message sent or a different message. Similarly the question must be asked for each message, whether it is necessarily sequenced, or whether some parallelism is allowed.

2.3 Detailed design

Detailed design focuses on developing the internal structure of each of the agents and how it will achieve its tasks within the system.

Detailed design is, by its nature, specific (at least to some degree) on the model of agents used; specifically on the internals of agents. In order to develop a detailed design process and associated models we need to select a particular agent model. We have chosen to focus on *plan-based* agents that use a library of user-defined plans. This model includes the various implementations of Belief, Desire, Intention (BDI) systems (e.g. PRS, dMARS, JAM, or JACK).

Although our detailed design caters well for BDI agents, it is not limited to them. The principles are easily adapted to the specifics of whichever development platform has been chosen, as long as it is within the broad general category of agents which use plans and react to events. The earlier phases of the methodology are *not* specific to a given agent model.

The focus of the detailed design phase is on defining capabilities (modules within the agent), internal events, plans and detailed data structures. A progressive refinement process is used which begins by describing agents' internals in terms of capabilities. The internal structure of each capability is then described, optionally using or introducing further capabilities. These are refined in turn until all capabilities have been defined. At the bottom level capabilities are defined in terms of plans, events, and data.

The functionalities from the specification phase provide a good initial set of capabilities, which can be further refined if desired. Sometimes there is also functionality akin to "library routines" which is required in multiple places - either within multiple agents, or within multiple capabilities within a single agent. Such functionality should also be extracted into a capability which can then be included into other capabilities or agents as required.

Capabilities are allowed to be nested within other capabilities and thus this model allows for arbitrarily many lay-

DRAFT

ers within the detailed design, in order to achieve an understandable complexity at each level.

Each capability should be described by a capability descriptor which contains information about the external interface to the capability - which events are **inputs** and which events are **produced** (as inputs to other capabilities). It also contains a natural language **description** of the functionality, a unique descriptive **name**, information regarding **interactions** with other capabilities, or **inclusions** of other capabilities, and a reference to data **read** and **written** by the capability. We use structured capability descriptor forms containing the above fields.

The agent overview diagram provides the top level view of the agent internals. It is very similar in style to the system overview diagram, but instead of agents within a system, it shows capabilities within an agent. This diagram shows the top level capabilities of the agent and the event or task flow between these capabilities, as well as data internal to the agent. By reading the relevant capability descriptors, together with the diagram, it is possible to obtain a clear high level view of how the modules within the agent will interact to achieve the overall tasks of the agent as described in the agent descriptor from the architectural design.

A further level of detail is provided by capability diagrams which take a single capability and describe its internals. At the bottom level these will contain plans, with events providing the connections between plans, just as they do between capabilities and between agents. At intermediate levels they may contain nested capabilities or a mixture of capabilities and plans. These diagrams are similar in style to the system overview and agent overview diagram, although plans are constrained to have a single incoming (triggering) event.

The final design artifacts required are the individual plan, event and data descriptors. These descriptions provide the details necessary to move into implementation. Exactly what are the appropriate details for these descriptors will depend on aspects of the implementation platform. For example if the context in which a plan type is to be used is split into two separate checks within the system being used (as is the case in JACK) then it is appropriate to specify these separately in the descriptor. Fields regarding what information an event carries assumes that events are composite objects able to carry information, and so on.

The plan descriptors we use provide an **identifier**, the **triggering event type**, **events, messages and actions**, the **plan steps** as well as a short natural language **description**, a **context** specification indicating when this plan should be used and a list of data **read** and **written**.

Event descriptors are used to fully specify all events, including those identified earlier. The event descriptor should identify the **purpose** of the event and any **data** that the event carries. We also indicate for each event its *coverage*: how many plans should be applicable to handle it? This is generally a range with the interesting values being zero, one, or many. Interesting cases include $1-1$ (always a single applicable plan), $1-N$ (there will always be a plan to handle the event and there may be more than one possible plan), $0-1$ (there will never be more than one applicable plan in a given situation, however it is possible for there to not be any applicable plan), and $0-N$. The coverage information is useful for debugging: a common programming error involves incorrect context conditions leading to the wrong plan being used, or to an event not being handled because no plan has is applicable.

Data descriptors should specify the fields and methods of any classes used for data storage within the system. If specialised data structures are provided for maintaining beliefs, these should also be specified.

An additional artifact that needs to be maintained as the design evolves is the **data dictionary**. The data dictionary should be started at the beginning of the project and developed further at each stage. The data dictionary is important in ensuring consistent use of names (for example, what is called "delivery info" in one place in the design should not be called "deliv. information" elsewhere).

3 Tool Support

In this section we describe the tool support for Prometheus that Agent Oriented Software and ourselves have developed. We begin by describing the support for the Prometheus methodology that is currently provided by the Jack Development Environment, including the recently-added design tool. We then describe further support that we have developed in a prototype tool and some ideas for supporting the specification phase that have not yet been implemented.

Our experience in using these tools has been positive. In particular, the cross checking provided by the prototype tool has been useful in finding inconsistencies in designs and in helping ensure that a design remains consistent as it is evolved.

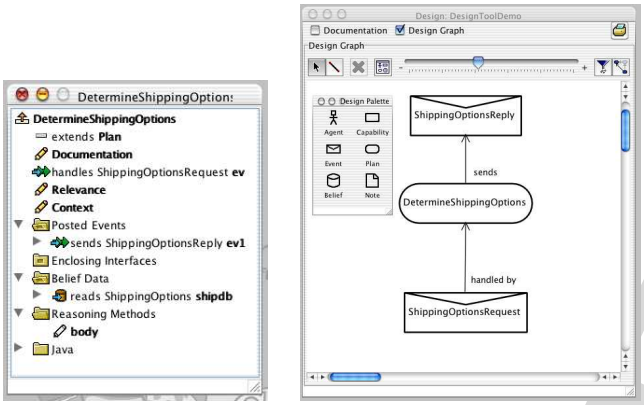
3.1 JACK Support for Prometheus

The Jack Development Environment (JDE) provides a graphical user interface that allows the structure[7] of an agent system to be built by drag-and-drop and by filling in forms (see below, left). The JDE supports the Prometheus methodology in that the concepts provided by JACK correspond to the artifacts developed in Prometheus' detailed

[7] Basically, everything except the bodies of plans. However, a graphical editor for plan bodies is under development.

DRAFT

design phase. It is important to realise that the agent structure described in the JDE generates JACK code that can be compiled and run.



A recent addition to the JDE is the *design tool*. This is a graphical canvas that allows overview diagrams to be drawn. However, unlike a drawing program, the diagrams are structured and are linked to the underlying agent system model. For example, the diagram above (right) was constructed by dragging two events and a plan onto the canvas. The links between the nodes are inserted (and labelled) automatically. If the diagram is changed (for example by adding a link between two nodes) then the entities involved are automatically updated. Likewise, if the entities are updated (for example, the plan is modified so that it sends a message rather than posts an event) then the diagram is automatically updated. This automatic insertion (and maintenance) of links ensures consistency between diagrams where possible.

Although consistency is maintained by the JDE in some areas, for example the JDE does not allow type errors to be made such as declaring a plan as handling a posted capability, there are still a number of places where an agent system design may become inconsistent. For example, since nodes are added to diagrams by hand, it is possible to create an agent overview diagram that accidently omits an event. It is possible for an agent or capability to declare that it handles a certain event without the agent/capability actually having a plan to handle the event.

Finally, the JDE and design tool do not (yet) support the earlier phases of the Prometheus methodology. For example, the process of deciding on agent types by combining functionalities, determining system functionality by developing use case scenarios and functionalities, and proceeding from scenarios via interaction diagrams to interaction protocols are not supported by the current tool. Some of these aspects are supported and addressed by the prototype tool described below.

Our experience has been that the design tool allows for the production of overview diagrams and that these are useful in understanding the structure and "big picture" of an agent system.

3.2 Prometheus Design Tool

The *Prometheus Design Tool* (PDT) is a prototype that we have developed. It allows a user to enter and edit a design, in terms of Prometheus concepts; check the design for a range of possible inconsistencies; automatically generate a range of diagrams according to the methodology; and automatically generate a design description (in LaTeX format) that includes descriptors for each design entity, a design dictionary, and the diagrams generated earlier.

PDT provides a graphical user interface (below) that allows a user to enter design elements (capabilities[8], events[9], agents, databases, and plans). Each of these has associated information that matches the Prometheus descriptors. For example, events have a coverage field, a sub-type field (e.g. action, percept, message, event) as well as a name and a description.



PDT supports the Prometheus methodology in a number of ways. It supports the process of deriving agent types from functionalities by deriving part of each agent's interface[10], by cross checking the declared interface of an agent against the functionalities that make up the agent type, and by generating coupling and acquaintance diagrams. It supports the process of developing the internals of agents in the detailed design phase by cross checking an agent's internals against the agent's declared interface, checking the consistency of a plan with its context, and by generating design diagrams at different levels (system overview, agent overview, and module overview).

We have found the cross checking to be very useful in maintaining the consistency of an evolving design. We have recently developed the design of the book store application in detail, working part-time over a period of weeks. When the design was first entered into PDT and cross-checked, a number of inconsistencies were detected. These included inconsistent naming of entities, and inconsistencies in sending and receiving messages (for example, the descriptor for

[8] We use this to represent both functionalities and capabilities.
[9] Encompassing percepts, actions, messages, and events.
[10] At the time of writing this has not yet been implemented.

DRAFT

functionality A would say it receives message m_1 from functionality B, whereas m_1 was actually being sent by functionality C).

As we have continued to develop the design the cross checking has proved to be valuable in ensuring that changes are made consistently across the design, for example, when we change a functionality by removing a message that it sends, the cross checking tells us where the message is expected and reminds us that we need to either modify the recipients by removing the expected message, or have another functionality post the message in question.

The cross-checking procedure consists of the following steps:

1. Check for undefined references and unused (unreachable) components.
2. Check for correct type and sub-type usage.
3. Check scenarios for consistency.
4. Check interface consistency of composite (see below) components.

We begin with a few definitions before discussing each of these in turn. The first two steps are straightforward and so we focus on the third and fourth.

Plans, events, and databases are *atomic* design components: they do not have children components. Agents and capabilities are *composite*: they have *children* that are either plans or capabilities. For the purposes of checking we also create a single implicit (composite) component of type *system* that has all of the defined agents as children.

Components of type system, agents, capabilities, and plans all have an *interface*: a definition of the events that are incoming and outgoing, as well as a list of databases that are read or written. For the purposes of cross-checking designs reading data is treated identically to an incoming event and writing data is treated identically to an outgoing event. We thus associate with each design component C the following attributes:

- A set of children components *children*(C), defined to be empty for atomic components.
- An interface, defined by two sets: the set of incoming events and read databases I_C, and the set of outgoing events and written databases O_C. The interface set is only defined for plans and for composite components.

Check for undefined references and unused (unreachable) components: It is clearly an error for a component to refer to an undefined component and this is checked in the obvious way. Although not an error, it is unusual for an component to be defined but not used. All components that are reachable from the system component by following children or interfaces are considered reachable. Warnings are issued for any unreachable components.

Check for correct type and sub-type usage: The following simple constraints are checked:

- Actions cannot appear in an incoming interface.
- Percepts cannot appear in an outgoing interface.
- The interface of a component (where defined) must contain only events and databases.
- The children of a (composite) component can only be capabilities or plans.
- The interface of the system component can only include actions (outgoing) or percepts (incoming).

Check scenarios for consistency:[11] As discussed in section 2.1 we use a fairly structured format for the steps in a scenario. This allows scenarios to be checked for internal consistency and for scenarios to be checked for consistency against functionalities (and to partially derive them).

We perform simple consistency checking on scenarios using the notions of *initiative* and *active participants*. Once an event is received, the functionality that has received it has the initiative. It can then perform actions/activities or send messages. Sending a message transfers the initiative to the recipient functionality. A warning is issued if a functionality acts in some way without having the initiative. The notion of active participants is similar to the notion of initiative, except that the active participant is a set that is extended by message sending and does not contract. It is an error for a functionality to act without it being in the set of active participants.

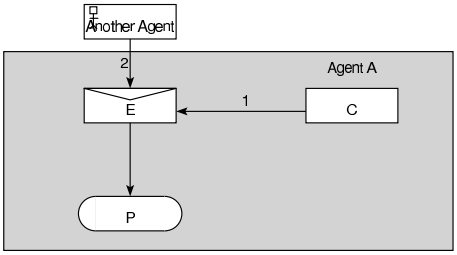
For example, if step 2 of the example scenario in section 2.1 was accidently omitted, then at step 3 Transport Information would be sending a message without being in the active participant set. This would be reported by the cross checking algorithm as an error. If, instead, step 3 was deleted, then at step 4 Online Interaction would be performing an activity without having the initiative, (although it is in the active participant set). This would be reported as a warning.

The interface of a functionality should be a superset of what is implied by its usage in scenarios. For example, where a scenario contains a step where an event is received by a functionality, then that event should be in the incoming interface of the functionality. Likewise, where a message is sent from functionality C_1 to C_2, then the message type should be in C_1's outgoing interface and in C_2's incoming interface. This constraint allows for functionalities to be partly derived from scenarios, and for them to be checked against scenarios.

[11] At the time of writing this has not been implemented.

DRAFT

Check interface consistency of composite components: Consider a composite component, say an agent A. Agent A has children (plans and capabilities) and each child has a defined interface. Any incoming event that appears in a child's interface must come from somewhere. It can either come from outside the agent, or from another[12] component that is internal to the agent (or both). For example, if A has as children a plan P and a capability C and P is triggered by the incoming event E, then either A has to allow for E to come in (from another agent), or E must be an *outgoing* event of C (or P). This situation is depicted below: one (or both) of the edges labelled 1 and 2 must be present, otherwise there is no source for the event E.



This condition can be defined formally as follows. Let C be a composite component and let I_C and O_C be its interface. We define the *combined incoming interface* of C's children $I_C^+ = \{i | ch \in children(C) \wedge i \in I_{ch}\}$ and the outgoing interface of C's children $O_C^+ = \{o | ch \in children(C) \wedge o \in O_{ch}\}$. Then the condition just says that any interface component (event or database) in a child's incoming interface (I_C^+) that is not in some child's outgoing interface (O_C^+) must be in the parent component's incoming interface (I_C). Hence I_C must include anything in I_C^+ that is not in O_C^+, thus $I_C^+ \setminus O_C^+ \subseteq I_C$ and conversely $O_C^+ \setminus I_C^+ \subseteq O_C$.

Another, simpler, condition is that any component that is in C's interface (where C is a composite component – that is, excluding plans) must be realised[13] in a plan contained in (a descendent of) C. This condition can be defined as $I_C \subseteq I_C^+$ and $O_C \subseteq O_C^+$. Putting these two conditions together we have that:

$$I_C^+ \setminus O_C^+ \subseteq I_C \subseteq I_C^+$$

$$O_C^+ \setminus I_C^+ \subseteq O_C \subseteq O_C^+$$

These do not determine the interface of a component, but only constrain it. The "looseness" corresponds to the case where an event can come from both the outside *and* from another child.

[12] Actually it can also come from the same component. For example, a plan can post an event that will trigger another instance of itself. This is just the agent equivalent of a recursive call.

[13] Only plans contain code that is executed, so if, for example, a capability reads a database or posts an event, then it must contain a plan that does these things.

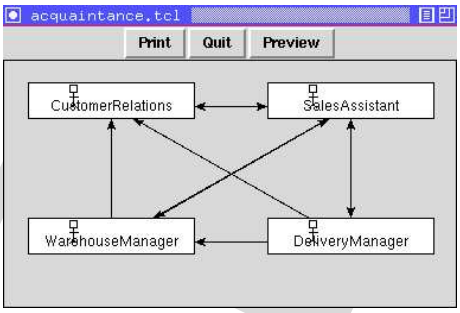


Figure 5. PDT Generated Diagram

A consequence of the definitions above is that an agent or capability *must* export any action or percept (since a percept/action cannot appear in both I_C^+ and O_C^+).

Once cross-checking has been performed the tool generates a number of design diagrams:

- System overview: all agents and their interfaces
- Agent/Capability: all capabilities/plans of the agent/capability and their interfaces
- Coupling: capabilities and exported data.
- Acquaintance: all agents and links between those agents that interact.

The diagram generation creates the diagram elements and links automatically. This is important for two reasons: firstly, it saves the designer having to manually create multiple diagrams; secondly, and more importantly, it means that mistakes such as leaving out edges or nodes cannot be made.

The placement of nodes (i.e. determining their (x, y) coordinates) is done manually by dragging them – links are automatically repositioned (figure 5). The diagrams positions are saved and encapsulated postscript is generated.

This encapsulated postscript is used in the generated design report. The report (in LaTeX format) includes a descriptor for each design element including automatically generated cross-referencing. Generated cross-referencing includes for agents and capabilities: what other components they interact with, and via which event/database; and for events and databases: what components read/post/write/receive them.

4 Related Work

There is currently a large amount of work being done in agent-oriented software engineering methodologies (e.g. [1, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 18, 19, 22], and see the references in [15]). It is not possible in a brief conference

DRAFT

paper to do justice to the significant amount of work being done.

The GAIA methodology [22] has, like Prometheus, been developed over a number of years by people experienced in building agent systems. However we found that the lack of a detailed design process - intentionally absent due to a desire for generality - meant that it did not provide sufficient support for the needs of those we were working with. There are similarities between Prometheus and Gaia for specification and architectural design. Our agent acquaintance diagrams are essentially the same as those used by Gaia, and the roles of Gaia are similar in concept to functionalities in Prometheus, although there are slightly different things which are considered.

The Tropos methodology does cover from early requirements to detailed design. Its detailed design is oriented very specifically towards JACK as an implementation platform. The tool support is currently only in the form of a diagram editor[14], rather than the consistency checking and automatic generation of some parts of the design that is part of PDT. The MaSE methodology [5] is one of the few methodologies that appears to have significant tool support.

Because the field is still young, none of the available methodologies can claim extensive use well beyond the group which has developed them. However the widespread development and use of these many new methodologies clearly indicates a need that is starting to be met, to provide more specific design methodologies for building agent systems.

Some approaches are based on taking UML[15] and extending or modifying it, as is done by Odell et al. [14] as well as others with a slightly different approach (e.g. [16]). This approach is sometimes justified by the observation that agents are just a special case of active objects.

In our experience, just extending UML did not provide sufficient assistance to start thinking in a different paradigm. Although agents can in some ways be seen as a specialised type of object, it is important to focus on such concepts as goals, plans and descriptions of situations. This is better supported by a more specialised methodolgy, borrowing and drawing from UML as appropriate.

In the current state-of-the-art, where the concepts and notations for designing agent systems are still not agreed, we believe it is best to consider possibilities without the world-view suggested by OO. Our students who just used modified OO design, had enormous difficulty in conceptualising agent applications and ultimately in building good agent systems.

Some of the significant differences between agent oriented design in Prometheus and OO methodologies are:

- The provision of a process for determining the agents in the system.
- Treating messages as components in their own right, not just as labels on arcs.
- Distinguishing percepts and actions from messages, and looking explicitly at percept processing.
- Distinguishing beliefs from agents: in OO both are (passive) objects.
- The identification of agent life-cycle issues.
- The use of protocols to capture the dynamics of agent interaction.
- The use of goals.

Although there are clear differences between Prometheus and OO methodologies, there are also commonalities. Although we do not believe that current OO methodologies are sufficient, we certainly *do* believe that they are relevant – agents are software, and indeed, many aspects of the Prometheus methodology have been based on OO methods and notations. For example, the scenarios are adapted from OO use-case scenarios; interaction diagrams are used as-is; AUML (itself an extension of UML) is used as-is, and Prometheus follows the RUP approach to applying an iterative process over clearly delineated phases.

In the longer term we see integrating agent methodologies with OO methodologies (and specifically with UML, since it is the de facto standard notation) as important steps in making the agent methodology accessible to developers. In this respect the work of [16] and [20] is valuable.

Certainly further work to compare and categorise the growing number of of proposed agent-oriented software engineering methodologies extending the work of [18] would be a valuable contribution.

5 Discussion and conclusions

We have briefly described the key aspects of the Prometheus methodology. The methodology has been in use for several years as a teaching tool, and has also been taught in industry workshops (most recently at the Australian AI conference, 2001). The feedback we have received indicates that it provides substantial guidance for the process of developing the design and for communicating the design within a work group. With student projects it is abundantly clear that the existence of the methodology is an enormous help in thinking about and deciding on the design issues, as well as conveying the design decisions.

[14] Conversation with Anna Perini at AAMAS'02 (July, 2002)

[15] Strictly speaking UML is not a methodology but rather a notation. However it is often coupled, either explicitly or implicitly with a methodology such as the Rational Unified Process (RUP).

DRAFT

There are a number of areas where the Prometheus methodology and PDT are currently being extended. These include extending PDT to support the specification phase and moving from scenarios to interaction protocols, representing and using life-cycle information and cardinality information (e.g. in debugging), testing techniques, and evaluating the methodology. Evaluating a software engineering methodology is difficult to do well, but is important.

We are also investigating how some of the design artifacts, such as the protocol definitions, and the capability diagrams, can be used for providing debugging and tracing support within the implemented system [17]. Having a design methodology which can be used through to testing and debugging is clearly advantageous in terms of an integrated and complete methodology.

Acknowledgements: We would like to acknowledge the support of Agent Oriented Software Pty. Ltd. and of the Australian Research Council (ARC) under grant CO0106934.

References

[1] F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *Int Journal of Cooperative Information Systems*, 6(1):67–94, 1997.

[2] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia, 1998.

[3] G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massonet. Agent oriented analysis using MESSAGE/UML. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 101–108, 2001.

[4] J. Debenham and B. Henderson-Sellers. Full lifecycle methodologies for agent-oriented systems – the extended OPEN process framework. In *Proceedings of Agent-Oriented Information Systems (AOIS-2002) at CAiSE'02*, Toronto, May 2002.

[5] S. A. DeLoach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.

[6] C. Iglesias, M. Garijo, and J. González. A survey of agent-oriented methodologies. In J. Müller, M. P. Singh, and A. S. Rao, editors, *ATAL-98*, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.

[7] N. Jennings and M. Wooldridge. Applications of intelligent agents. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, chapter 1, pages 3–28. Springer, 1998.

[8] N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[9] E. A. Kendall, M. T. Malkoun, and C. H. Jiang. A methodology for developing agent based systems. In C. Zhang and D. Lukose, editors, *First Australian Workshop on Distributed Artifi cial Intelligence*, 1995.

[10] D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. In *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96). LNAI 1193*. Springer-Verlag, 1996.

[11] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In R. van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996.

[12] J. Lind. A development method for multiagent systems. In *Cybernetics and Systems: Proceedings of the 15th European Meeting on Cybernetics and Systems Research, Symposium "From Agent Theory to Agent Implementation"*, 2000.

[13] J. Mylopoulos, J. Castro, and M. Kolp. Tropos: Toward agent-oriented information systems engineering. In *Second International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS2000)*, June 2000.

[14] J. Odell, H. Parunak, and B. Bauer. Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artifi cial Intelligence.*, 2000.

[15] L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In *Third International Workshop on Agent-Oriented Software Engineering*, July 2002.

[16] M. Papasimeon and C. Heinze. Extending the UML for designing JACK agents. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*, Aug. 2001.

[17] D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'02)*, 2002.

[18] O. Shehory and A. Sturm. Evaluation of modeling techniques for agent-based systems. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 624–631. ACM Press, May 2001.

[19] L. Z. Varga, N. R. Jennings, and D. Cockburn. Integrating intelligent systems into a cooperating community for electricity distribution management. *Int Journal of Expert Systems with Applications*, 7(4):563–579, 1994.

[20] G. Wagner. A UML profi le for external AOR models. In *Third International Workshop on Agent-Oriented Software Engineering*, July 2002.

[21] M. Winikoff, L. Padgham, and J. Harland. Simplifying the development of intelligent agents. In M. Stumptner, D. Corbett, and M. Brooks, editors, *AI2001: Advances in Artifi cial Intelligence. 14th Australian Joint Conference on Artifi cial Intelligence*, pages 555–568. Springer, LNAI 2256, Dec. 2001.

[22] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.

DRAFT

Simplifying the Development of Intelligent Agents

Michael Winikoff, Lin Padgham, and James Harland

RMIT University, Melbourne, AUSTRALIA
{winikoff,linpa,jah}@cs.rmit.edu.au,
http://www.cs.rmit.edu.au/~{winikoff,linpa,jah}

Abstract. Intelligent agents is a powerful Artificial Intelligence technology which shows considerable promise as a new paradigm for mainstream software development. However, despite their promise, intelligent agents are still scarce in the market place. A key reason for this is that developing intelligent agent software requires significant training and skill: a typical developer or undergraduate struggles to develop good agent systems using the Belief Desire Intention (BDI) model (or similar models). This paper identifies the concept set which we have found to be important in developing intelligent agent systems and the relationships between these concepts. This concept set was developed with the intention of being clearer, simpler, and easier to use than current approaches. We also describe briefly a (very simplified) example from one of the projects we have worked on (RoboRescue), illustrating the way in which these concepts are important in designing and developing intelligent software agents.
Keywords: AI Architectures, distributed AI, multiagent systems, reactive control, software agents.

1 Introduction

Intelligent agents is a powerful Artificial Intelligence technology which shows considerable promise as a new paradigm for mainstream software development. Agents offer new ways of abstraction, decomposition, and organisation that fit well with our natural view of the world and agent oriented programming is often considered a natural successor to object oriented programming [6]. It has the potential to change the way we design, visualise, and build software in that agents can naturally model "actors" – real world entities that can show autonomy and proactiveness. Additionally, social agents naturally model (human) organisations ranging from business structure & processes to military command structures. A number of significant applications utilising agent technology [5] have already been developed, many of which are decidedly non-trivial.

An intelligent agent is one which is able to make rational decisions, i.e., blending proactiveness and reactiveness, showing rational commitment to decisions made, and exhibiting flexibility in the face of an uncertain and changing environment.

Despite their promise, intelligent agents are still scarce in the market place [1]. There is a real technical reason for this: developing intelligent agent software currently requires significant training and skill. Our experience (and the experience of others) is that a

[1] Although abuse of buzzwords is, alas, all too common.

typical developer or final-year undergraduate student struggles to develop good agent systems using the Belief Desire Intention (BDI) model (or similar models).

Decker [3] discusses problems that undergraduate students have in approaching agent oriented development. These include a lack of suitable background in AI (planning and goal oriented programs), poor software engineering skills, and a lack of experience at dealing with concurrent and distributed programming/debugging, and with communication protocols.

We have found the key problem to be that students cannot clearly identify the necessary pieces to break the program into and thus tend to build monolithic plans which try to handle all contingencies internally, rather than create an appropriate collection of plans which can be applied in different contexts. They also have significant difficulty with interfacing the agent to its environment. Other reasons why developing intelligent agent systems is difficult include:

- Immature tool support: There is a lack of good debugging tools and of tools which integrate an internal agent architecture with suitable middleware & infrastructure. Additionally, many tools are research prototypes and lack efficiency, portability, documentation, and/or support.
- The need for processes and methodologies: Programmers are familiar with designing object oriented systems. However, the design of agent oriented systems differs in a number of ways. For example, identifying roles, goals, and interaction patterns.
- Design guidelines and examples: Designing a collection of plans to achieve a goal is different to designing a single procedure to perform a function. This difference is fundamental – developing intelligent agents is a different programming paradigm and needs to be learnt and taught as such.
- Complex concepts such as intentions are difficult to explain; this isn't helped by a lack of agreement on concepts and inconsistent terminology.
- Lack of a suitable[2] text book: much of the work on intelligent agents is scattered across many research papers (sometimes collected into volumes).

In the process of working on a number of agent programs, teaching students and assisting them to build agent programs, and developing and running workshops for academia and industry[3], we have developed an initial process of agent design and development for BDI systems.[4] This process is explained more fully in the technical report [8] and is still being refined and developed.

This paper identifies the concept set which we have found to be important in developing intelligent agent systems and the relationships between these concepts. These of course rely heavily on the standard Belief Desire Intention (BDI) concepts [9, 10] though we have found it necessary to clarify some of the differences between just what these concepts are in the initial philosophical work [1], the logical theories [2, 9, 13] and the implementations such as PRS, dMars and JACK. We have also found it important

[2] Suitable: Aimed at undergraduates or professional developer and contains enough detail to answer the question "How would I actually go about building an intelligent agent?".

[3] Workshops have been developed and delivered in association with Agent Oriented Software

[4] Using primarily dMars from the Australian Artificial Intelligence Institute, and more recently JACK from Agent Oriented Software.

to place some emphasis on the concepts of percepts and actions which appear in many generic models of agents (e.g. [12]) and which are very important in the interfacing of the agent deliberation to the external environment. We have found a need to separate more clearly between events and goals than is done in dMars or JACK and to provide greater support within the execution engine for reasoning about goals than is usually done in BDI agent systems [14].

Our work in general is focussed on multi-agent systems although this paper concentrates on the concepts required for the internals of each intelligent agent - a necessary pre-requisite for multi-agent systems with teams or societies containing such agents.

2 Background: The BDI Model

The BDI model [9, 10] is a popular model for intelligent agents. It has its basis in philosophy [1] and offers a *logical theory* which defines the mental attitudes of Belief, Desire, and Intention using a modal logic; a *system architecture*; a *number of implementations of this architecture* (e.g. PRS, JAM, dMars, JACK); and *applications* demonstrating the viability of the model. The central concepts in the BDI model are:

Beliefs: Information about the environment; *informative*.
Desires: Objectives to be accomplished, possibly with each objective's associated priority/payoff; *motivational*.
Intentions: The currently chosen course of action; *deliberative*.
Plans: Means of achieving certain future world states. Intuitively, plans are an abstract specification of both the means for achieving certain desires and the options available to the agent. Each plan has (i) a body describing the primitive actions or subgoals that have to be achieved for plan execution to be successful; (ii) an invocation condition which specifies the triggering event[5], and (iii) a context condition which specifies the situation in which the plan is applicable.

The BDI model has developed over about 15 years and there are certainly strong relationships between the theoretical work and implemented systems. The paper [10] describes an abstract architecture which is instantiated in systems such as dMars and JACK and shows how that is related to the BDI logic. However, the concepts we have found useful for development within these systems do not necessarily match the concepts most developed in the theoretical work. Neither are they necessarily exactly the concepts which have arisen within particular implemented systems such as JACK. An additional complication is small differences between similar concepts, such as Desires and Goals, which receive differing emphasis in different work at different times.

Desires are understood to be things the agent wants to achieve. They play an important role in the philosophical foundations, but the logical theory deals primarily with Goals, which are assumed to be a consistent set of desires. At the implementation level the motivational concept is reduced to events – goals are implicit and the creation of a new goal is treated as an event which can trigger plans. Events are ignored in the theoretical framework although they play a key role in implementations. In the theoretical model plans are simply beliefs or intentions. However in the implementations

[5] Some events are considered as goal-events.

plans are a central concept. Some key differences between the philosophy, theory, and implementation viewpoints of BDI are shown in the table below.

Philosophy:	Belief	Desire	Intention
Theory:	Belief	Goal	Intention
Implementation:	Relational DB (or arbitrary object)	Event	Running Plan

3 Concepts for Intelligent Agents

We describe the set of concepts which we have come to use in developing intelligent agent applications. We believe that these are necessary and sufficient for building the sort of applications appropriately approached using BDI agents, and we hope that they are simple, and clearly explained. The work to develop a formal semantic framework for these concepts, thus developing closer links between a theoretical framework and an implemented development platform, is work in progress.

We build up our description of an intelligent agent by beginning with a basic, and universally agreed upon (see for example [12]), property of agents: they are situated (see figure 1). Thus, we have **actions** and **percepts**. Internally, the agent is making a **decision**: from the set of possible actions As it is selecting an action (or actions) to perform ($a \in As$). Loosely speaking, where the description of the agent's internal workings contains a statement of the form "[select] $X \in Y$" then we have a decision being made. Thus the type of decisions being made depend on the *internal agent architecture*.

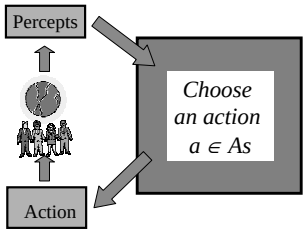


Fig. 1. Agents are Situated

An **action** is something which an agent does, such as *move_north* or *squirt*. Agents are situated, and an action is basically an agent's ability to effect its environment. In their simplest form actions are atomic and instantaneous and either fail or succeed. In the more general case actions can be durational (encompassing behaviours over time) and can produce partial effects; for example a failed *move_to* action may well have changed the agent's location. In addition to actions which directly affect the agent's environment, we also want to consider "internal actions". These correspond to an ability which the agent has which isn't structured in terms of plans and goals. Typically, the ability is a piece of code which either already exists or would not benefit from being written using agent concepts, for example image processing in a vision sub-system.

A **percept** is an input from the environment, such as the location of a fire and an indication of its intensity. The agent may also obtain information about the environment through sensing actions.

A **decision**: The essence of intelligent agents is rational decision making. There are a number of generic, non-application-specific questions which intelligent agents must answer, such as: *Which* action shall I perform now? *Which* goal do I work on now? *How* shall I attempt to realise this goal? *Where* shall I go now (for mobile agents)? And *who* shall I interact with (for social agents)?

Mechanisms to answer these kinds of questions are core intelligent agent processes. They result in decisions which must fulfil rationality conditions, in that we expect that decisions be persistent and only be revisited when there is a good reason for doing so. It is also important that the answer to the questions not be trivial: if an agent only has a single goal at a time and a single means of realising this goal then we have reduced the agent to the special case of a conventional program and there is no scope for decision making or for flexible, intelligent behaviour.

Note that although the concept of a decision is fundamental to intelligent agents, it is not always necessary to represent the decisions explicitly. For example, the decision regarding choice of goal could be represented using a "current goal" variable which is updated when a decision is made.

We now consider the internal workings of the agent (see figure 2). We want our intelligent agents to be both *proactive* and *reactive*. A proactive agent is one which pursues an agenda over time. The agent's proactiveness implies the use of **goals** and modifies the agent's internal execution cycle: rather than select an action one at a time, we select a goal which is persistent and constrains our selection of actions. A *reactive* agent is one which will change its behaviour in response to changes in the environment. An important aspect in decision making is balancing proactive and reactive aspects. On the one hand we want the agent to stick with its goals by default, on the other hand we want it to take changes in the environment into account. The key to reconciling these aspects, thus making agents suitably reactive, is to identify *significant* changes in the environment. These are **events**. We distinguish between percepts and events: an event is an interpreted percept which has significance to the agent. For example, seeing a fire is a percept. This percept could give rise to a *new fire* event or a *fire under control* event depending on history and possibly other factors.

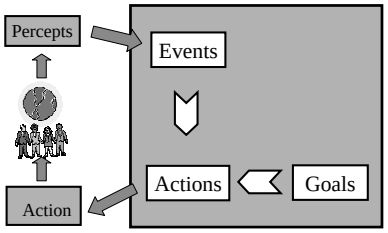


Fig. 2. Proactive Agents have Goals, Reactive Agents have Events

A **goal** (variously called "task", "objective", "aim", or "desire") is something the agent is working on or towards, for example *extinguish_fire*, or *rescue_civilian*. Often goals are defined as states of the world which the agent wants to bring about; however, this definition rules out maintenance goals (e.g. "maintain cruising altitude") and avoidance goals, or safety constraints (e.g. "never move the table while the robot is drilling"). Goals give the agent its autonomy and proactiveness. An important aspect of proactiveness is the persistence of goals: if a plan for achieving a goal fails then the agent will consider alternative plans for achieving the goal in question. We have found that goals require greater emphasis than is typically found in existing systems. It is important for the developer to identify the top level goals of the agent as well as subsidiary goals which are used in achieving main goals. Our modified execution engine does significantly more reasoning about goals than is usual in BDI implementations [14], including reasoning about interference between goals and how to select goals when it is not consistent to pursue them simultaneously. We differentiate between top level goals and

subsidiary goals in that subsidiary goals are not important in their own right and may therefore be treated differently in the reasoning process than top-level goals.

An **event** is a significant occurrence. Events are often extracted from percepts, although they may be generated internally by the agent, for example on the basis of a clock. An event can trigger new goals, cause changes in information about the environment, and/or cause actions to be performed immediately. Actions generated directly by events correspond to "reflexive" actions, executed without deliberation. Events are important in creating reactive agents in that they identify important changes which the agent needs to react to.

Agents in realistic applications usually have limited computational resources and limited ability to sense their environment. Thus the auxiliary concepts of **plan** and **belief** are needed. Beliefs are effectively a cache for perceived information about the environment, and plans are effectively a cache for ways of pursuing goals (see figure 3). Although both of these concepts are "merely" aids in efficiency, they are not optional. Beliefs are essential since an agent has limited sensory ability and also it needs to build up its knowledge of the world over time. Plans are essential for two reasons. The first is pure computational efficiency: although planning technology and computational speed are improving, planning from action descriptions is still incompatible with real time decision making. The second reason is that by providing a library of plans we avoid the need to specify each action's preconditions and effects: all we need to provide for an action is the means to perform it. This is significant in that representing the effects of continuous actions operating over time and space in an uncertain world in sufficient detail for first principles planning is unrealistic for large applications.

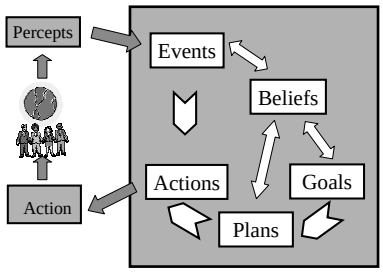


Fig. 3. Adding Plans and Beliefs

A **plan** is a way of realising a goal, for example a plan for achieving the goal *extinguish_fire* might specify the three steps: plan a route to the fire, follow the route to the fire, and squirt the fire until it has been put out. Although the concept of a plan is common there is no agreement on the details. From our point of view it is not necessary to adopt a specific notion of a plan, rather we can specify abstractly that a plan for achieving a goal provides a function which returns the next action to be performed. This function takes into account the current state of the world (beliefs), what actions have already been performed, and might involve sub-goals and further plans. For computational reasons it is desirable for this to at least include a "library of recipes" approach, rather than requiring construction of plans at runtime from action descriptions.

A **belief** is some aspect of the agent's knowledge or information about the environment, self or other agents. For example an agent might believe there is a fire at X because she saw it recently, even if she cannot see it now.

These concepts (actions, percepts, decisions, goals, events, plans, and beliefs) are related to each other via the **execution cycle** of the agent. An agent's execution cycle

follows a sense-think-act cycle, since the agent is situated. The think part of the cycle involves rational decision making, consisting of the following steps: (depicted in figure 4)

1. Percepts are interpreted (using beliefs) to give events
2. Beliefs are updated with new information from percepts
3. Events yield reflexive actions and/or new goals
4. Goals are updated, including current, new and completed goals.
5. If there is no selected plan for the current goal, or if the plan has failed, or if reconsideration of the plan is required (due to an event) then a plan is chosen.
6. The chosen plan is expanded to yield an action
7. Action(s) are scheduled and performed

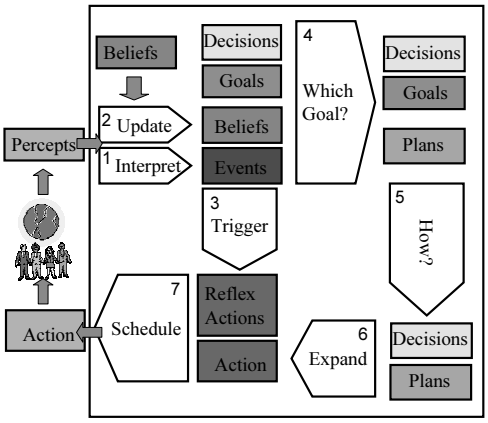


Fig. 4. Agent Execution Cycle

By comparison, the BDI abstract execution cycle [10] consists of the following steps: (1) use events to trigger matching plans (options), (2) select a subset of the options, (3) update the intentions and execute them, (4) get external events, and (5) drop successful and impossible attitudes. The execution cycle presented here differs from the BDI execution cycle in a number of ways including the use of reflexive actions, the derivation of events by interpreting percepts, the process of going from goals to plans to actions, and increased reasoning about goals. However, the two primary contributions are the role of top level goals (which are distinguished from events and from sub-goals, and are persistent) in achieving proactiveness, and the role of events (as *significant* occurrences) in creating suitably reactive agents.

4 A Case Study: RoboRescue

One of the applications we have worked on recently is RoboRescue. We describe a greatly simplified version of a part of this application in order to illustrate concretely the concepts we have identified.

RoboRescue [11] is a long-term (50 years!) project which has the goal of creating robotic squads which could be deployed in the aftermath of a disaster such as an earthquake. Tasks to be carried out include rescuing & evacuating people and controlling fires. Challenges faced include the lack of information and an environment which contains obstacles (including collapsed buildings, obstructed roads, fires, etc.) and potentially limited communication.

The RoboRescue simulator has a number of components which simulate different aspects of a disaster scenario. The intention is that these aspects combine synergistically to provide a realistically complex and challenging environment. There are a number of

different types of agents in the system including fire engines, ambulances, civilians, and police agents. At each simulator cycle agents receive visual information and possibly heard information.

Due to space limitations in this paper we focus on a single agent type (fire engine) and a simple set of its behaviours focussed on fire extinguishing.

To design the fire engine agent using the concepts described earlier we look at each concept and identify instances of it in the agent system. This process is, in general, iterative: when designing a plan we may realise that the agent needs to know a certain piece of information which implies the addition of a belief which might imply the addition (or modification) of goals and plans. A comprehensive methodology for the detailed design of agent systems is described in [8]. Here we concentrate on illustrating the way in which instances of the relevant concepts are identified and defined.

Decisions are not considered here since questions which an agent needs to answer as it runs aren't specific to a given application domain; rather, they are specific to a given agent architecture. Detailed design regarding how agents will work together, exactly what plans will be used and what sub-goals will be needed is a design task well beyond the scope of this paper and requiring the more extensive methodology of [8]. Rather, we focus on the initial aspects of the process and the identification of some of the relevant concepts in each class. More concept instances will inevitably be identified as the design progresses.

Percepts: Percepts represent an interface to the environment, so are often, as in this case, at least partially predefined. There are two types of percepts in RoboRescue - visual and auditory. Auditory information may be broadcast information which in the case of a fire-engine agent may be a message from another fire-engine agent, the fire-engine center, or a civilian either crying for help or stating that they have heard a cry for help. It may also be a message directed specifically to that agent from another fire-engine agent. The content of messages from fire-engine agents and the fire-engine center is an aspect of perceptual information which is under the control of the designer and must be decided. Visual information contains a current view of the environment including such things as roads, buildings, fires and their intensity, etc.

Percepts must be processed to build up knowledge of the environment (beliefs) and to extract events. In the case of the visual information in RoboRescue it is first processed in order to add any information to the map of the world that is built incrementally. It is then processed to extract the position of each fire which is assessed to see whether there is a fire-related event which should be generated and to further update the knowledge of the environment.

Actions: The actions which an agent can perform are also part of its interface to the environment. As indicated earlier, there may be additional actions defined beyond those that affect the environment, but these are a starting point. In this case the agent can perform the external actions of *squirting* which reduces the fire at the current location, *moving* which moves the agent an unspecified number of steps along a given route, *telling* (broadcasting) and *saying* (sending) a message. The move and squirt actions may need to be applied repeatedly to achieve the desired goal, for example a fire may need to be squirted multiple times before it is extinguished and the agent may need to move several times before it reaches its destination.

We also identified early on the action of planning a route between two points as an internal action. A route is a necessary parameter for the move action, code existed for planning a route given the map, a current position and a destination, and there seemed to be no clear advantage in using goals and plans to achieve this task, particularly given the existing code. Thus we had an initial set of five actions, four that were part of the external interface to the simulator and one which was internal.

Goals: An obvious major goal of the fire engine agent is to put out any fire. We also identified a goal of discovering fires. Additional thought yielded for us the further goals of assisting a team-mate to put out a fire and coordinating a team effort to extinguish a fire. The goals of the agent obviously have to do with the motivation for the system, but are not externally defined in the way that at least some of the percepts and actions are. Choosing the appropriate set of top-level goals is one of the early design decisions that need to be made.

It is tempting to treat goals as being implied by the beliefs: any belief in the existence of a fire implies a goal extinguish the fire. However, this approach has a number of issues. Firstly, it is hard to indicate that certain fires should not be pursued (e.g. because another agent is dealing with them). Secondly, it is difficult to add goals which are not directly prompted by environmental cues. For example, the goal to find fires exists independent of cues in the environment, and should result in exploratory behaviour if no other goals are inhibiting this goal.

In addition to identifying the goals of an agent, we need to specify when to adopt and drop goals (including how to recognise when they are achieved), plans for achieving these goals, sub-goals that may be part of achieving each goal and relative priorities and interactions between the goals.

As indicated in section 3, events, i.e. significant things which happen in the environment, often result in the adoption of goals. Thus the event of receiving a message from a team-mate requesting help in extinguishing a fire is likely to result in a goal to assist that team mate to put out his fire. The event of noticing a fire is likely to result in a goal to extinguish that fire. Thus we move onto events.

Events: Here we are identifying significant occurrences that are likely to make the agent add or delete goals, change goal priorities, or change how the agent is pursuing a goal. We also look for significant occurrences which affect our beliefs. Many events will be the result of processing percepts - e.g. extracting information about fires and their locations, checking these against a list of fires we already know about, and obtaining any “new-fire” events. Some events will also be generated internally as a result of the agent’s own behaviour. For example after sending a request for assistance with a fire, the agent may generate a “help-requested” event. As events are used to update beliefs, trigger plans and generate reflexive actions there are likely to be a large set of events which are developed iteratively in the process of developing the full design. The detailed design methodology in [8] allows for a layered identification of events in conjunction with incremental refinement of sets of plans.[6]

The initial events that we identify typically have to do with the significant occurrences that will alert the agent to the need to instantiate one of its top level goals, recognise one of its top level goals as achieved, or indicate a need for reconsideration.

[6] These sets of plans and related events are actually JACK capabilities.

In this example such events include

- "new-fire" which causes instantiation of an "extinguish" goal and also causes a reflexive action to broadcast the existence of the fire to other fire-engine agents;
- "fire-extinguished" event which causes the agent to recognise that a goal has been achieved;
- "fire-urgent" event which indicates fire is growing and is larger than a given threshold. This indicates a need for reconsideration and in our design may lead to a goal to co-ordinate a team effort to extinguish the fire;
- "help-requested" which leads to a reconsideration of current priorities and may lead to a goal to assist team-mates;

Identification of events determines what interpretation or processing we need to do with the percepts received in order to be able to recognise events. This in turn also often affects what information about the environment we need to represent - e.g. to be able to generate a new fire event from a visual percept we need to explicitly represent which fires we already know about.

Beliefs: Beliefs are really any knowledge the agent maintains. Some of this information may be kept in the form of a special purpose knowledge database, other information may be kept in arbitrary suitable data structures. For this application the primary information needed is an internal map of the environment including the location of fires, roads, buildings, etc. Updating of this data structure is part of the important processing of percepts.

Beliefs tend to be used primarily in two ways. The first is in extracting events from percepts: to recognise a new fire we have to have knowledge about existing fires. The second is in determining which plan should be used to achieve a goal in a particular situation. Information that allows us to choose between two alternative ways of achieving a goal - or even whether there is any way of achieving a goal, depends on some representation of beliefs.

Some of the beliefs we have identified as important here (in addition to the map) are: which fires are being attended to and by who; current priority of fires; and whether a route is available to a particular location (it may not be, either due to blockages, or insufficient information about the environment).

Plans: Plans describe various ways for us to achieve our goals. To get the full power of the BDI approach it is advantageous to define simple plans, with use of sub-goals wherever possible. Initially there may only be one straightforward way to achieve each goal, but new variations can be added in a modular fashion. This allows development of a simple agent that manages straightforward cases first, with addition of variations for more complex cases afterwards. Plan sets need to be checked for coverage and overlap regarding the situations which can arise.

A very simple pair of plans for achieving the goal to extinguish a fire at X would be one plan which simply squirts until the fire is extinguished (suitable if the agent is already at location X), and another plan which obtains a route to X, moves to X, then squirts until the fire is extinguished (suitable if the agent is not already at X). Alternatively we could have a plan for putting out a fire and a plan for extinguishing a fire as shown below:

Goal: put-out_fire(position)	**Goal:** extinguish_fire(position)
plan_route(position)	**Condition:** location = position
move(route)	squirt
extinguish_fire(position)	extinguish_fire(position) **unless** nofire(position)

In fact this plan is simple enough that it can be implemented reactively using a set of trigger response rules, all that needs to be stored for this approach is the target's coordinates. However this approach is unable to handle sequences of actions where the triggers aren't in the environment and is unable to manage commitment. For these reasons plans need internal state to track what has been done, and need to be able to specify a sequence of actions to be done.

5 Discussion

We have presented the concepts we have found important in building BDI agent systems and the relationships between these concepts. We have found these concepts to be clearer and easier to teach and use than the BDI model. The concepts presented:

- Distinguish between goals and sub-goals, between percepts and events, and between events and goals. BDI implementations, by comparison, do not distinguish these and merge them all into an "event" type. We feel that this distinction is important since goals and events play roles in achieving reactivity and proactiveness and have rather different properties: for example, goals persist until they are achieved.
- Explicitly represent goals. This is vital in order to enable selection between competing goals, dealing with conflicting goals, and correctly handling goals which cannot be pursued at the time they are created and must be delayed [14].
- Highlight goal selection as an important issue. By contrast, BDI systems simply assume the existence of a selection function.
- Emphasise the importance of percepts and actions.
- Highlight the role of events in creating reactive agents.

We also introduced reflex actions, generalised the concept of a plan, decomposed the concept of intentions (which we have found to be difficult to explain and teach) into the simpler notion of a decision, and provided a hierarchical, staged presentation of the concepts.

Our design process (which is still being refined and developed, and which is explained more fully in [8]) is focussed on the detailed design and we view it as complementary to methodologies such as GAIA [15] and Tropos [7] which focus more on the higher level design and analysis and on the requirements aspects of agent systems.

The choice of concepts was driven by three sources. Primarily, our experience working on a number of agent programs, teaching students and assisting them to build agent programs, and developing and running workshops for academia and industry. Secondly, a survey of a range of agent systems (see below); and finally, "bottom up" derivation of concepts from first principles.

Although there is some degree of consensus in the deliberative agent research community that the BDI model is a reasonable common foundation for intelligent agents

there are also known shortcomings of the model [4]. Thus, in order to avoid being "BDI-biased" we surveyed a range of agent systems which address the internals of an intelligent software agent. For more details see *http://www.cs.rmit.edu.au/~winikoff/SAC*.

There is much further work to be done including continuing to apply the concepts and design process to various applications. We are also starting to survey students and professionals regarding the concepts they regard as natural for developing agent systems. Developing a formal semantics for the concepts we have identified, as well as developing support tools (design, development, and debugging) are also high priorities. Finally, the concepts identified need to be extended (and revised) to support the creation of social intelligent agents.

Acknowledgements: We would like to acknowledge the support of Agent Oriented Software Pty. Ltd. and of the ARC (under grant CO0106934).

References

1. M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.
2. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
3. K. Decker. DECAF update: Agents for undergrads & planning for a smart scheduler. Presentation at the Workshop on Infrastructure for Agents, MAS, and scalable MAS at Autonomous Agents 2001.
4. M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In ATAL, 1999.
5. N. Jennings and M. Wooldridge. Applications of intelligent agents. Chapter 1 in *Agent Technology: Foundations, Applications, and Markets*. Springer, 1998.
6. N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.
7. J. Mylopoulos, J. Castro, and M. Kolp. Tropos: Toward agent-oriented information systems engineering. In *Second International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS2000)*, June 2000.
8. L. Padgham and M. Winikoff. A methodology for agent oriented software design. Technical Report TR-01-2, School of Computer Science and Information Technology, RMIT University, 2001.
9. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-Architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning, Proceedings of the Second International Conference*, pages 473–484, Apr. 1991.
10. A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 439–449, San Mateo, CA, 1992. Morgan Kaufmann Publishers.
11. RoboCup-Rescue home page. *http://robomec.cs.kobe-u.ac.jp/robocup-rescue/*, 2001.
12. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
13. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
14. J. Thangarajah. Representation of goals in the belief-desire-intention model, 2000. Honours thesis, supervised by Lin Padgham.
15. M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.

Goals and Functionalities

Identify the system goals of the application and the functionalities needed.

Goals:

……..

……..

……..

……..

……..

……..

……..

……..

……..

……..

Functionalities: ***Associated goal #***

……..

……..

……..

……..

……..

……..

……..

……..

……..

……..

……..

Functionality Descriptor

High-level design description of each functionality.

Name:

……..

Description:(2-3 sentences)

……..

……..

……..

……..

……..

Percepts/Triggers:

……..

……..

……..

Actions:

……..

……..

……..

Messages sent:

……..

……..

……..

Data used:

……..

……..

……..

Data produced:

……..

……..

……..

Interactions:

……..

……..

Scenario # _____

Short Name: _____

Description:

……

……

……

……

context:

……

……

Scenario steps: (PERCEPT/ACTION/message/activity with associated functionalities and data

Step Number	Step Description	Functionality(ies)	Data read	Data written
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….
…….	…….	…….	…….	…….

……expand as necessary☐

Variations on main case:

……..

……..

……..

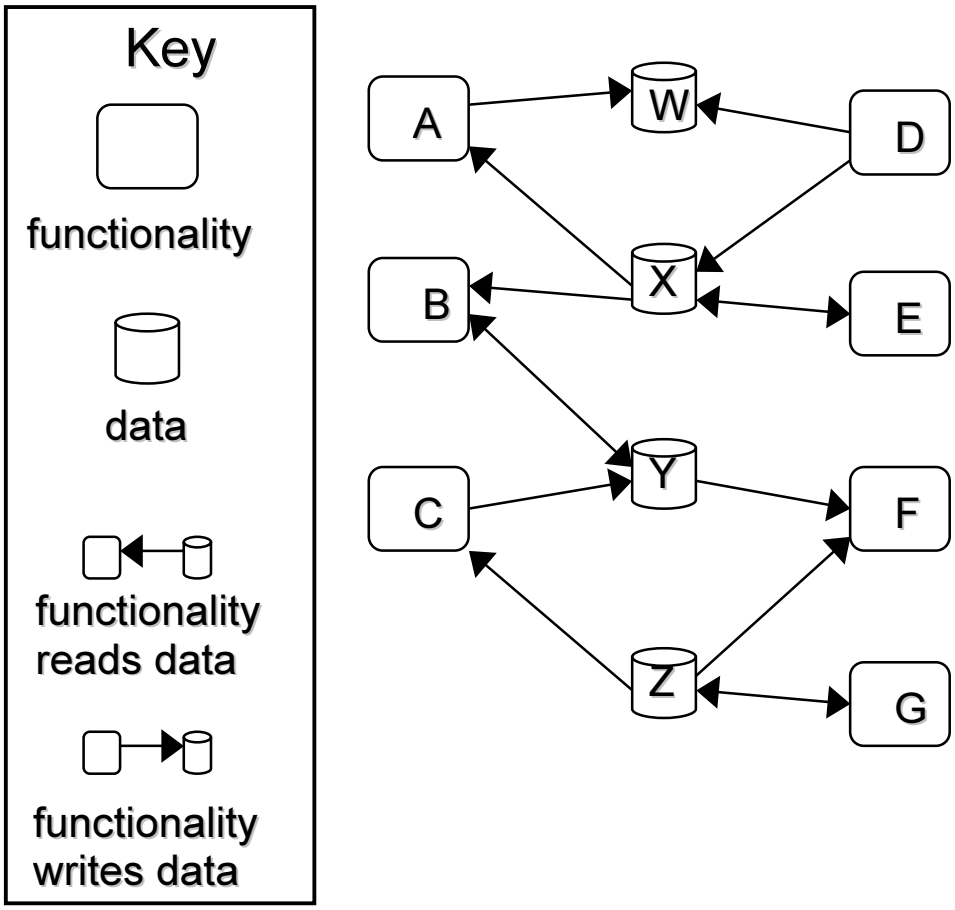
……..

……..

…… expand as necessary

Data Coupling Diagrams

Use functionality descriptors to develop data coupling diagrams to assist in identifying possible agent groupings.

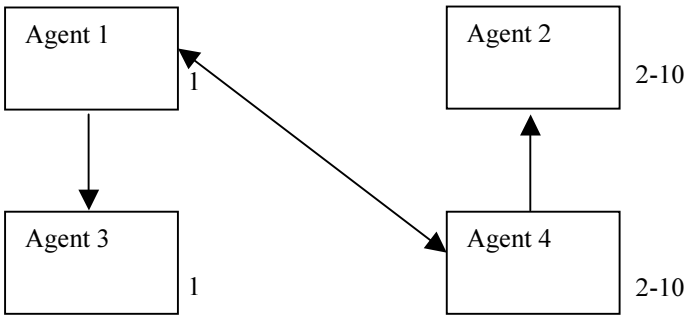


Agent Acquaintance Diagram

Indicate which functionalities will be grouped together in a single agent type and give an agent level overview of the system showing:

1) The types (classes) of agents
2) The expected number of each type of agent
3) The interaction between agents of different types

Agent 1 (functionalities 1 and 2)
Agent 2 (functionality 3)
Agent 3 (functionality 4)
Agent 4 (functionalities 5 and 6)



Communication

2-10 Number of agents of this type

Percept and Action Descriptors

Percept Descriptor

Name:

……..

Description:

……..

……..

Source: (sensor or other source)

……..

……..

Structure: (in what form does the data arrive)

……..

……..

Important events from percept:

……..

……..

Auxiliary data: (data other than the percept, needed for generating events, e.g. previous percepts)

……..

……..

Action Descriptor

Name:

……..

Description:

……..

……

Parameters: (affecting performance of action, e.g. speed for move action)

……..

……..

Temporality: (instantaneous/durational)

……..

……..

Failure detection: (when might it fail and how would you know)

……..

……..

Partial change:

……..

……..

Side effects:

……..

……..

Message Descriptor

Name:

……..

Description:

……..

……..

……..

Information carried:

……..

……..

……..

Protocol / conversation it belongs to:

……..

……..

From agent:

……..

……..

……..

To agent:

……..

……..

……..……

Purpose: (Trigger response, update information, etc.)

……..

……..

……..……

Agent Type Descriptors

Description:

……..

……..

Goals:

……..

……..

Number of instances: (e.g. singleton, one per xxx)

……..

……..

Capabilities needed:

……..

……..

……..

Agent lifetime:

……..

……..

Initialisation:

……..

……..

Demise:

……..

……..

Percepts:

……..

Actions:

……..

Messages sent:

……..

……..

Messages received:

……..

……..

Interactions:

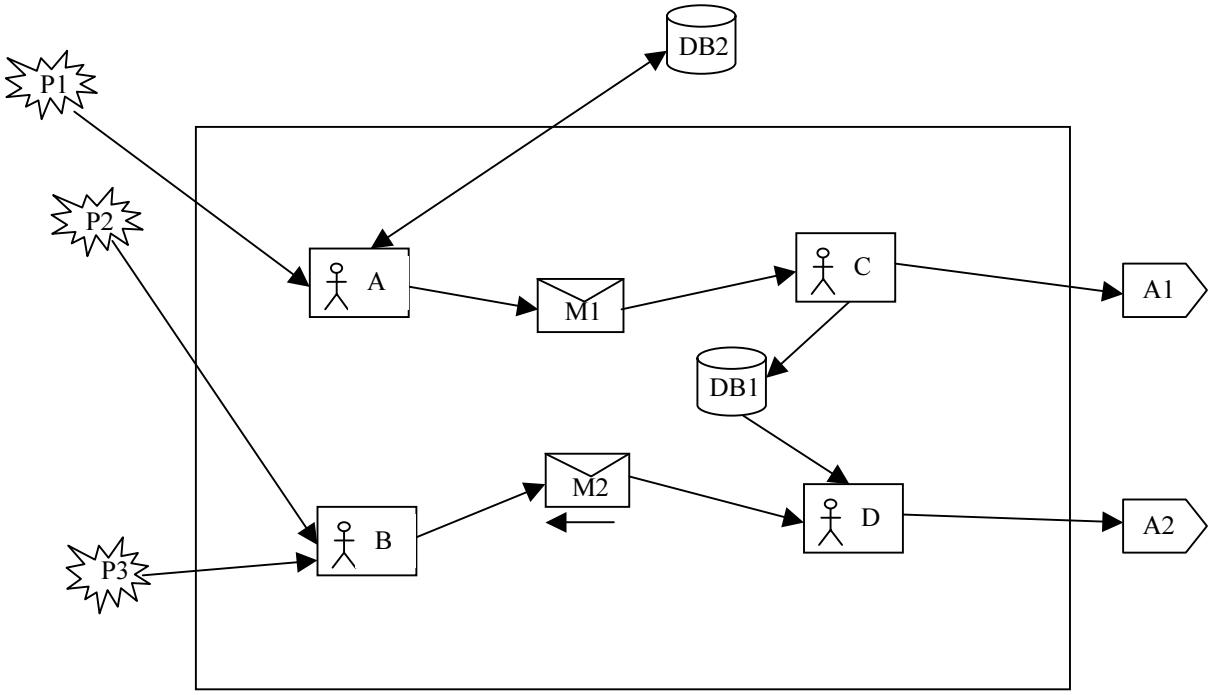
……..

……..

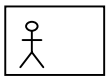
Information read	*Information written*
……	……
……	……
……	……

System Overview Diagram

The system overview diagram shows the agents, the percepts the agents respond to, messages (or protocols/conversations) between agents, actions of agents and use of external or shared data stores. It can also show interfaces with non-agent parts of the system or other sub-systems.



KEY


Agent


Message


Message with reply


Percept


Database

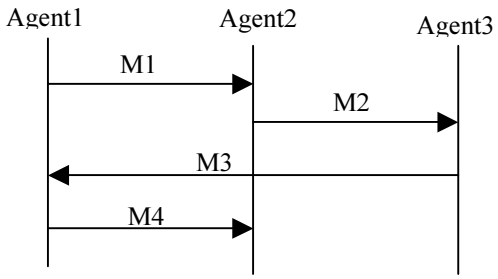

Action


Protocol


Protocol with reply

Interaction Diagrams and Protocols

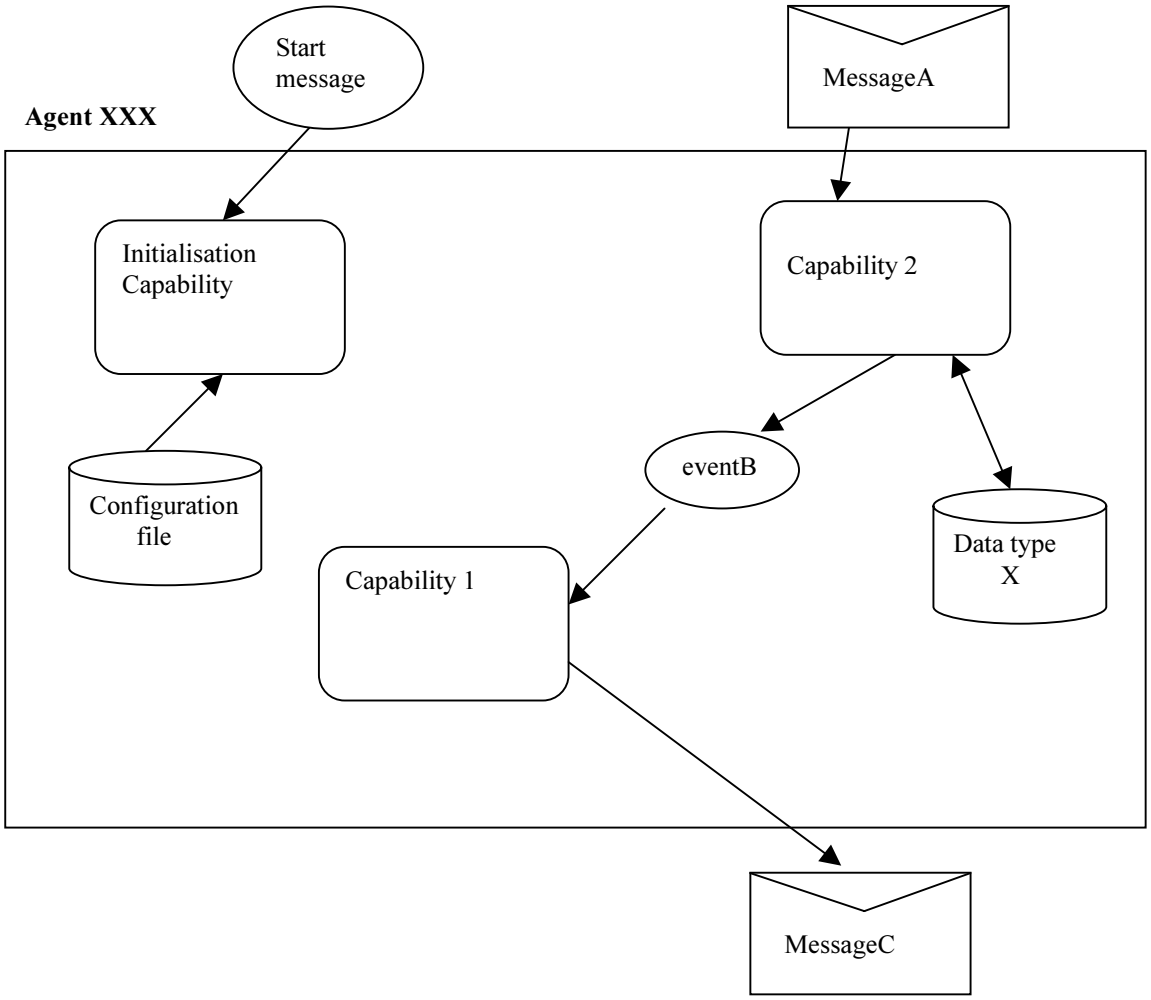
Use scenarios and information about which functionalities belong to which agents, to develop interaction diagrams. This will often suggest modifications to the scenario to achieve simplification. Generalise interaction diagrams to generate protocols or specifications of legitimate conversations.



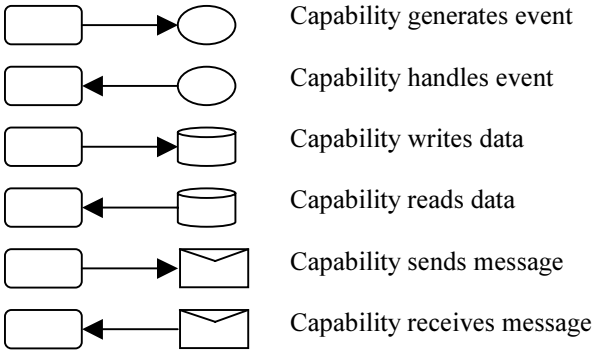
Agent Overview Diagram

Each agent may have a number of ***capabilities***. Capabilities are building blocks which can be re-used between agents. To extend the functionality of a capability, you can include it within another capability.

The interface between capabilities is via the events that they handle and generate. For each agent show an agent overview indicating the capabilities of the agent and the event generation and handling of each capability. The agent overview diagram should also show data important for that agent, and which capabilities use each data source.



Key



Capability Descriptor

Name:

……..

Description:

……..

……..

……..

……..

Interactions with other capabilities:

……..

……..

……..

……..

Events handled:

……..

……..

……..

Events generated:

……..

……..

……..

Included capabilities:

……..

……..

Information used	*Information provided*
……	……
……	……
……	……
……	……
……	……
……	……

Capability Overview (Data-Event-Plan DEP) Diagram

Detailed solution design. Illustrates the data-event-plan relationships within each capability

Documentation:

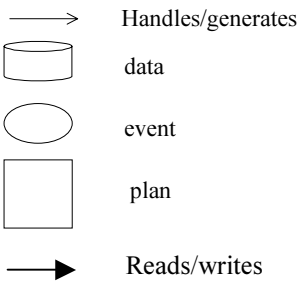
……

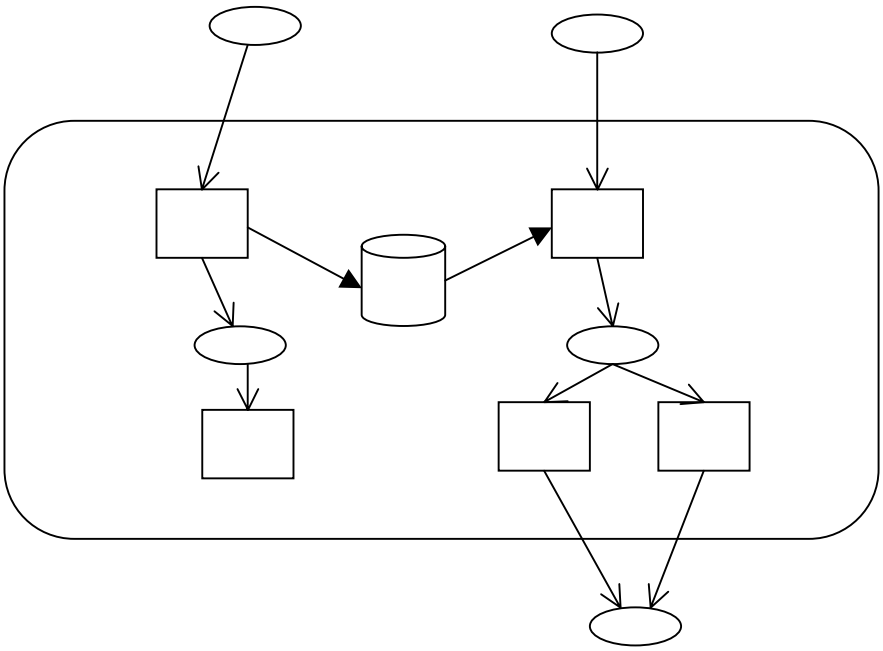
……

……

……

……





Event Type Descriptor

Detail design description of an event type.

Name:

……..

Purpose:

……..

……..

……..

……

……..

Arguments:

……..

……..

……..

……..

……..

……..

Coverage: Is there always an applicable plan to handle this event. If not explain exceptions.

……..

……..

……..

……..

Overlap: Will there be multiple plan types that are applicable in some situations. If so explain.
Will there be multiple plan instances of the same type that are applicable in some situations?

……..

……..

……..

……..

Plan Type Descriptor

Detail design description of a plan type.

Documentation:

……..

……..

……..

Event type handled:

……..

Context:

……..

Failure: (when/how might this plan fail)

……..

……..

Fail method: (what needs to be done if this plan fails, cleanup etc.)

……..

……..

Subtasks generated:

……..

……..

Message events sent:

……..

……..

New goals generated:

……..

……..

Procedure:

……..

……..

……..

……..

……..

……..

……..

Information read	*Information written*
……	……
……	……
……	……

Beliefs and Data Structures

Detail design description of a belief structure.
This may be a specialised structure, depending on the implementation platform, or a programming language data structure, or a database. Data should be designed and specified using whatever paradigms and diagrams are appropriate – e.g. class diagrams, or database tables.

……..

……..

……..

……..

Design Dictionary for _____

project / application / module

List of Names for each type.

Agents:

Capabilities:

Plans:

Actions:

Events:

Messages:

Percepts:

Data fields:

Resources

Agents at RMIT:
http://www.cs.rmit.edu.au/agents

SAC (Simplifying Agent Concepts) Project and Prometheus
http://www.cs.rmit.edu.au/agents/SAC

Agent Oriented Software, Pty. Ltd.
http://www.agent-software.com

UMBC AgentWeb
http://agents.umbc.edu/
Includes AgentNews: an email newsletter
http://agents.umbc.edu/agentnews/current/

AgentLink
http://www.agentlink.org
Includes AgentLink Newsletter
http://www.agentlink.org/newsletter/index.html

“Intelligent Agents”, Michael Wooldridge.
In G. Weiss, editor: Multiagent Systems, The MIT Press, April 1999.
(Downloadable from *http://www.csc.liv.ac.uk/~mjw/pubs/*)

Conference on Autonomous Agents and Multiagent Systems
2003: Melbourne, Australia
http://www.aamas-conference.org

Journal on Autonomous Agents and Multi-Agent Systems
http://www.wkap.nl/prod/j/1387-2532