Individual lecture presentation

Kate Slezavina

Content

- **“Prometheus: A Methodology for Developing Intelligent Agents”**

  Lin Padgham and Michael Winikoff RMIT University, GPO Box 2476V, Melbourne, AUSTRALIA

  This paper presents the *Prometheus* methodology for developing intelligent agent systems.

- **“Multiagent systems engineering (MaSE) ”**

  SCOTT A. DELOACH, MARK F. WOOD AND CLINT H. SPARKMAN *Air Force Institute of Technology Graduate School of Engineering and Management Department of Electrical and Computer Engineering Wright-Patterson Air Force Base, OH 45433-7765*

  This paper describes the MaSE methodology for developing heterogeneous multiagent systems.

- **“Comparing Agent Oriented Methodologies”**

  Khanh Hoa Dam Michael Winikoff School of Computer Science and Information Technology RMIT University, Melbourne, Australia

  This paper presents a comparison of two agent-oriented methodologies: MaSE and Prometheus.

Prometheus

- The claim is that Prometheus is developed in sufficient detail to be used by a **nonexpert**.

Prometheus

- A detailed comparison with the existing methodologies:
- Supports the development of *intelligent* agents which use goals, beliefs, plans, and events. By contrast, many other methodologies (i.e. MASE) treat agents as "simple software processes that interact with each other to meet an overall system goal"

Prometheus

- Provides “start-to-end” support (from specification to detailed design and implementation) and a *detailed process*.

Prometheus

- 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.

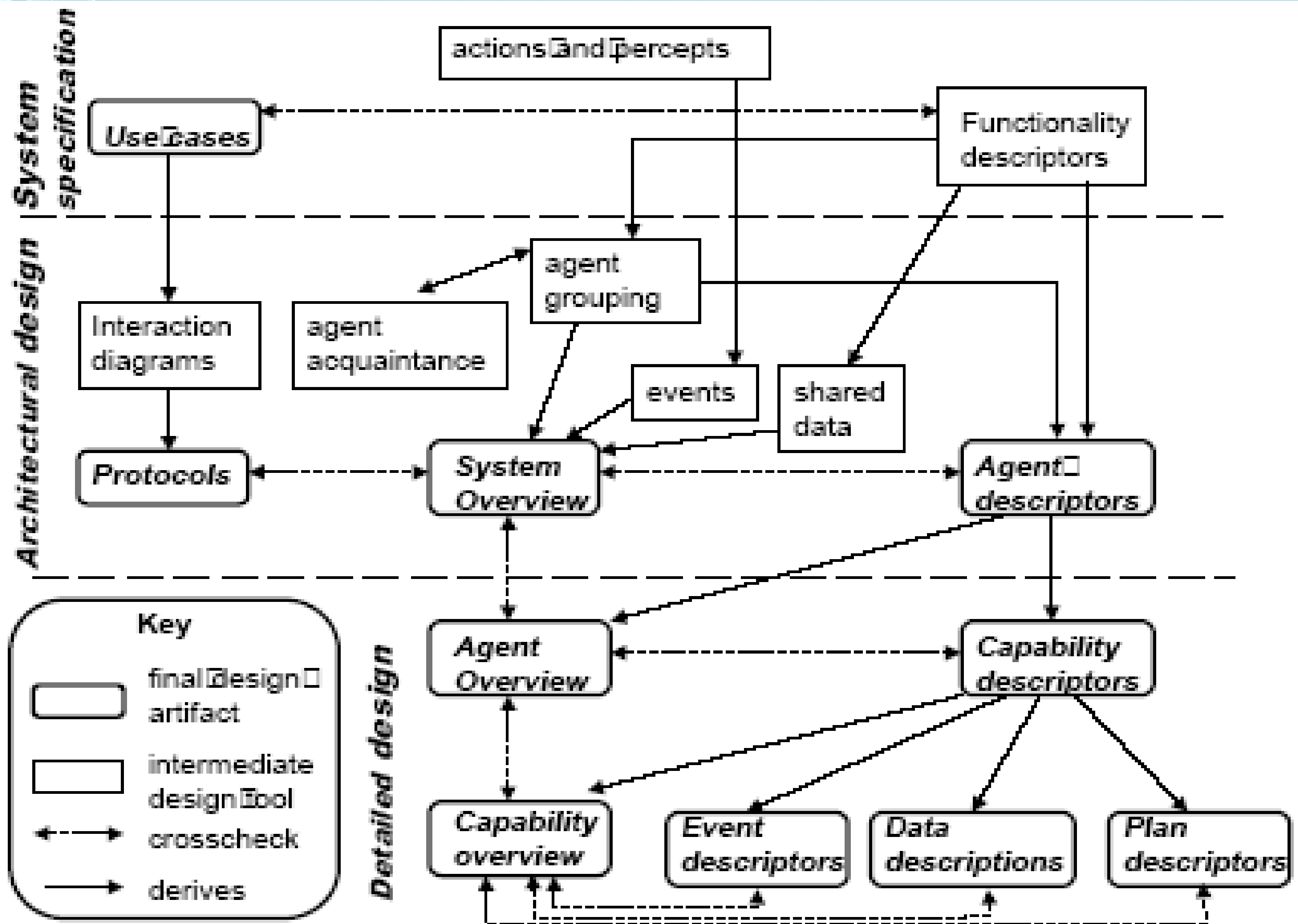
Prometheus

- 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.

Prometheus

- 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.

Prometheus



Prometheus (*system specification phase*)

System specification phase

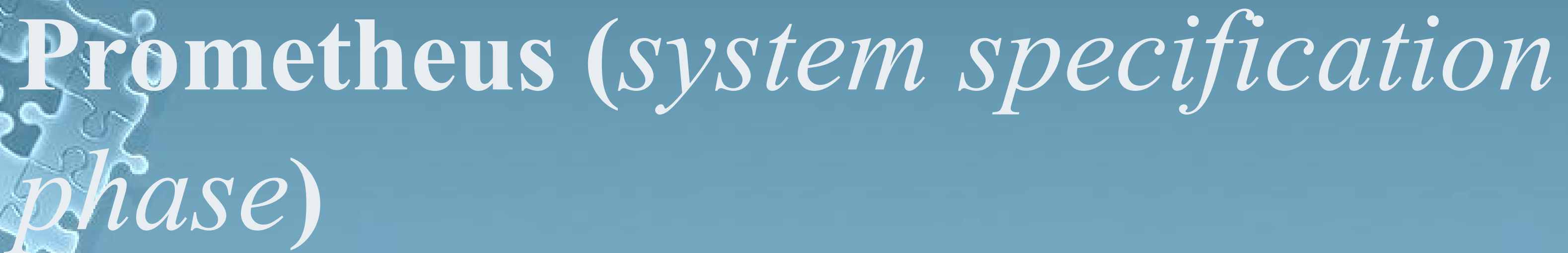
Agent systems are typically situated in a changing and dynamic environment, which can be affected, through not totally controlled by the agent system.

One of the earliest questions which must be answered is how the agent system is going to interact with this environment.

We will call incoming information from the environment "percepts", and the mechanisms for affecting the environment "actions".

An event is a significant occurrence for the agent system, whereas a percept is raw data available to the agent system.

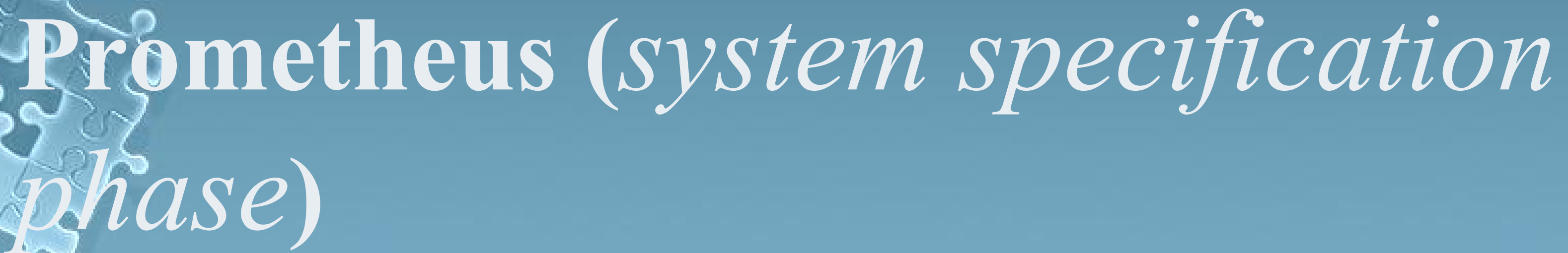
EXAMPLE: The online bookstore has the percepts of customers visiting the website, selecting items, placing orders (using forms), and receiving email from customers, delivery services and book suppliers. Actions are bank transactions, sending email, and placing delivery orders.



Prometheus (*system specification phase*)

In parallel with discovering or specifying the percepts and actions the developer must start to describe what it is the agent system should do in a broader sense - the functionalities of the system.

In defining a functionality it is important to also define the information that is required, and the information 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.



Prometheus (*system specification phase*)

While functionalities focus on particular aspects of the system, *use case scenarios* give a more common view of the system.

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. The use case templates 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**.

Architectural design

The major decision to be made during the architectural design is which agents should exist. We assign functionalities to agents by analyzing the artifacts of the previous phase to suggest possible assignments of functionalities to agents. The process of identifying agents by grouping functionalities involves analyzing the reasons for and against groupings of particular functionalities. If functionalities use the same data it is an indication for grouping 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.

Prometheus (*Architectural design*)

- 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.

Prometheus (*Architectural design*)

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.

Prometheus (*Architectural design*)

Questions which need to be resolved about agents at this stage include:

How many agents of this type will there be? 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 initialization - what needs to be done?

What data does this agent need to keep track of?

What events will this agent react to?

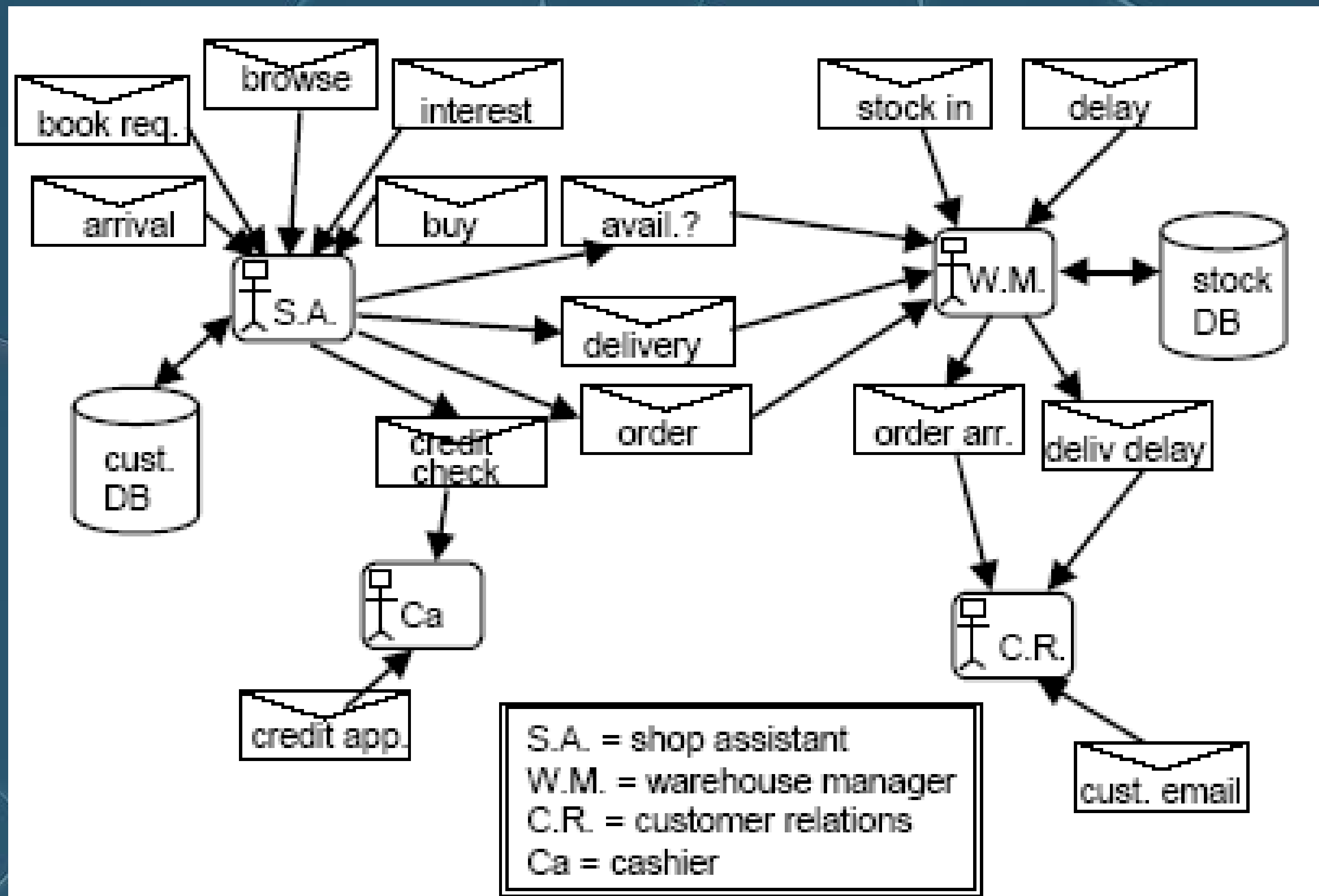
Prometheus (*Architectural design*)

- 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.
- **Shared data objects** (if any) must also be identified at this stage. A good design will minimize those, but there may be situations where it is reasonable to have shared data objects. Data objects should be specified using traditional object oriented techniques.

Prometheus (*Architectural design*)

- The **system overview diagram** events and shared data objects. It is definitely the single most important artifact of the entire design process, although of course it cannot really be understood fully in isolation. 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.

Prometheus (*Architectural design*)

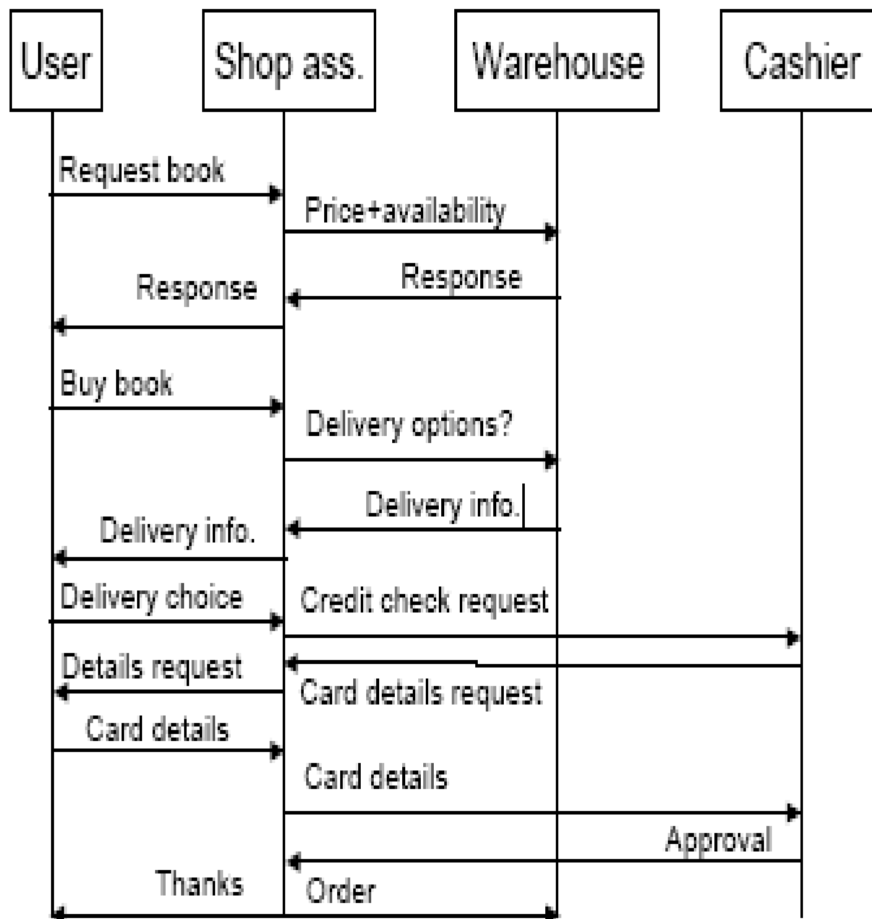


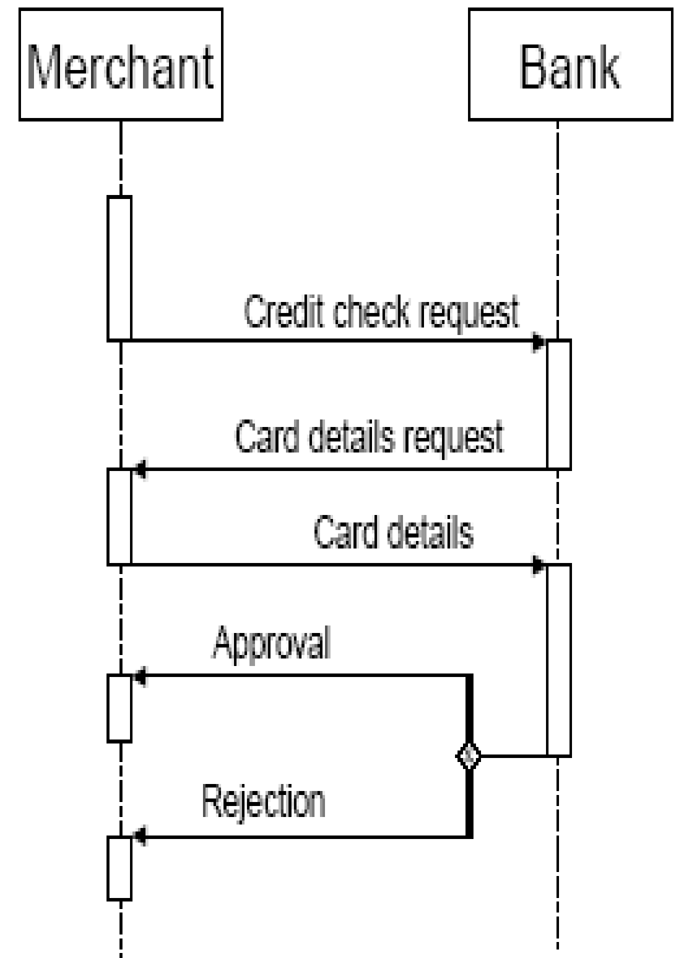
Prometheus (*Architectural design*)

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.

Interaction diagrams, like use cases, give only a partial picture of the system’s behavior. 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. Next figure (right) shows the protocol for the credit check portion of the interaction diagram shown in this figure (left).

Prometheus





- ***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

The focus of the detailed design phase is on defining capabilities (modules within the agent), internal events, plans and detailed data structures. The internal structure of each capability is then described, optionally using or introducing further capabilities. At the bottom level capabilities are defined in terms of plans, events, and data.

Prometheus (*Detailed design*)

- 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.

- 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.

- 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.

Prometheus (*Detailed design*)

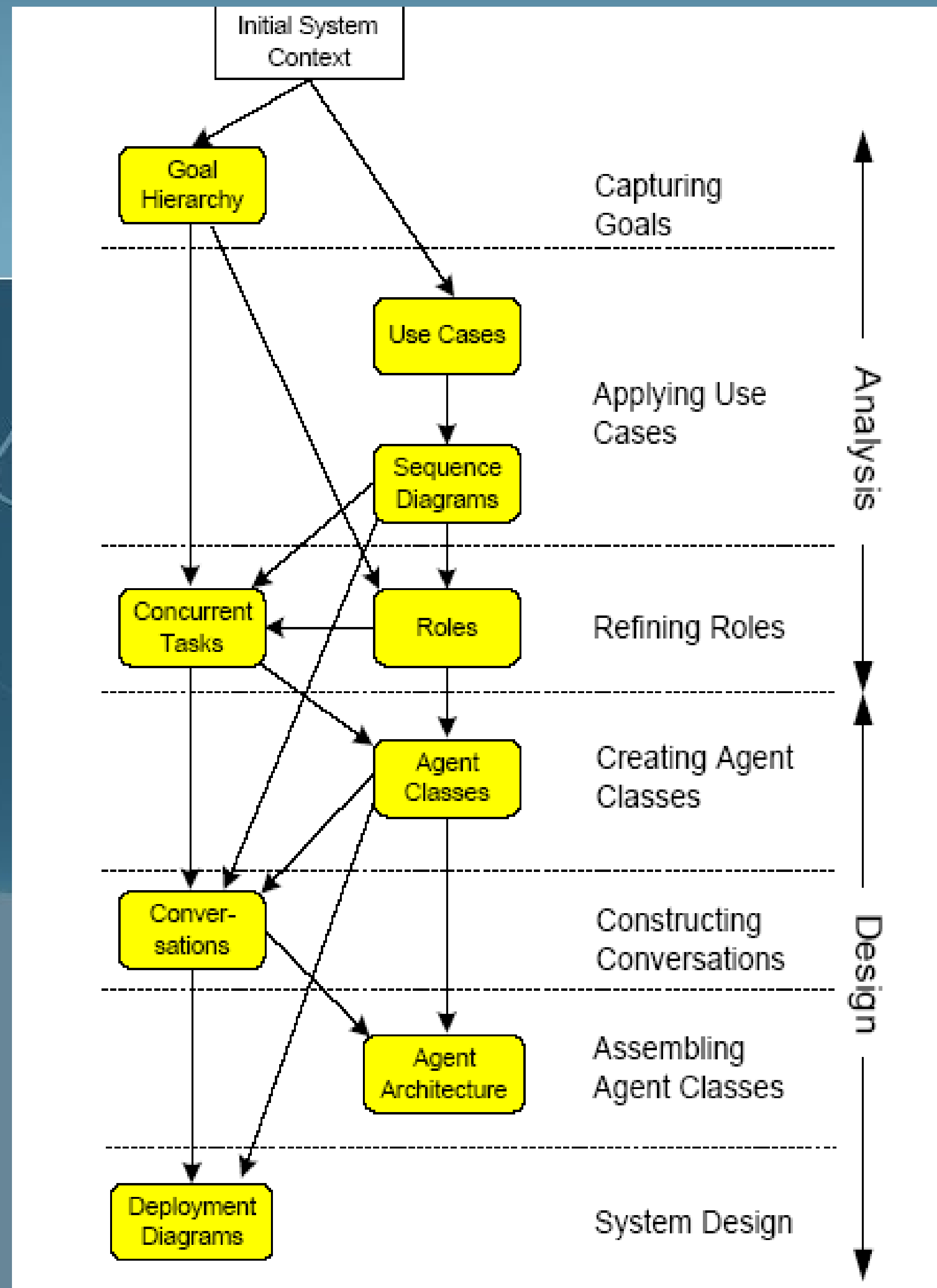
- One of the advantages of this methodology is the number of places where automated tools can be used for consistency checking across the various artifacts of the design process. For example, the input and output events for an agent must be the same on the system overview diagram and on the agent overview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

MaSE

- MaSE uses the abstraction provided by multiagent systems for developing intelligent, distributed software systems. MaSE is a further abstraction of the object-oriented paradigm where agents are a specialization of objects. Instead of simple objects, with methods that can be invoked by other objects, agents coordinate with each other via conversations and act proactively to accomplish individual and system-wide goals.

MaSE

- The general operation of MaSE follows the phases and steps shown on the right side:



MaSE

- The MaSE **Analysis phase** consists of three steps: Capturing Goals, Applying Use Cases, and Refining Roles.
- The **Design phase** has four steps: Creating Agent Classes, Constructing Conversations, Assembling Agent Classes, and System Design.

MaSE

A major strength of MaSE is the ability to track changes throughout the process.

Every object created during the analysis and design phases can be traced forward or backward through the different steps to other related objects. For instance, a goal derived in the Capturing Goals step can be traced to a specific role, task, and agent class.

Analysis Phase

The purpose of the MaSE Analysis phase is to produce a set of roles whose tasks describe what the system has to do to meet its overall requirements. A role describes an entity that performs some function within the system. In MaSE, each role is responsible for achieving, or helping to achieve specific system goals or sub-goals. MaSE roles are analogous to roles played by actors in a play or by members of a typical company structure.

The overall approach in the MaSE Analysis phase is fairly simple. Define the system goals from a set of functional requirements and then define the roles necessary to meet those goals. While a direct mapping from goals to roles is possible, MaSE suggests the use of Use Cases to help validate the system goals and derive an initial set of roles.

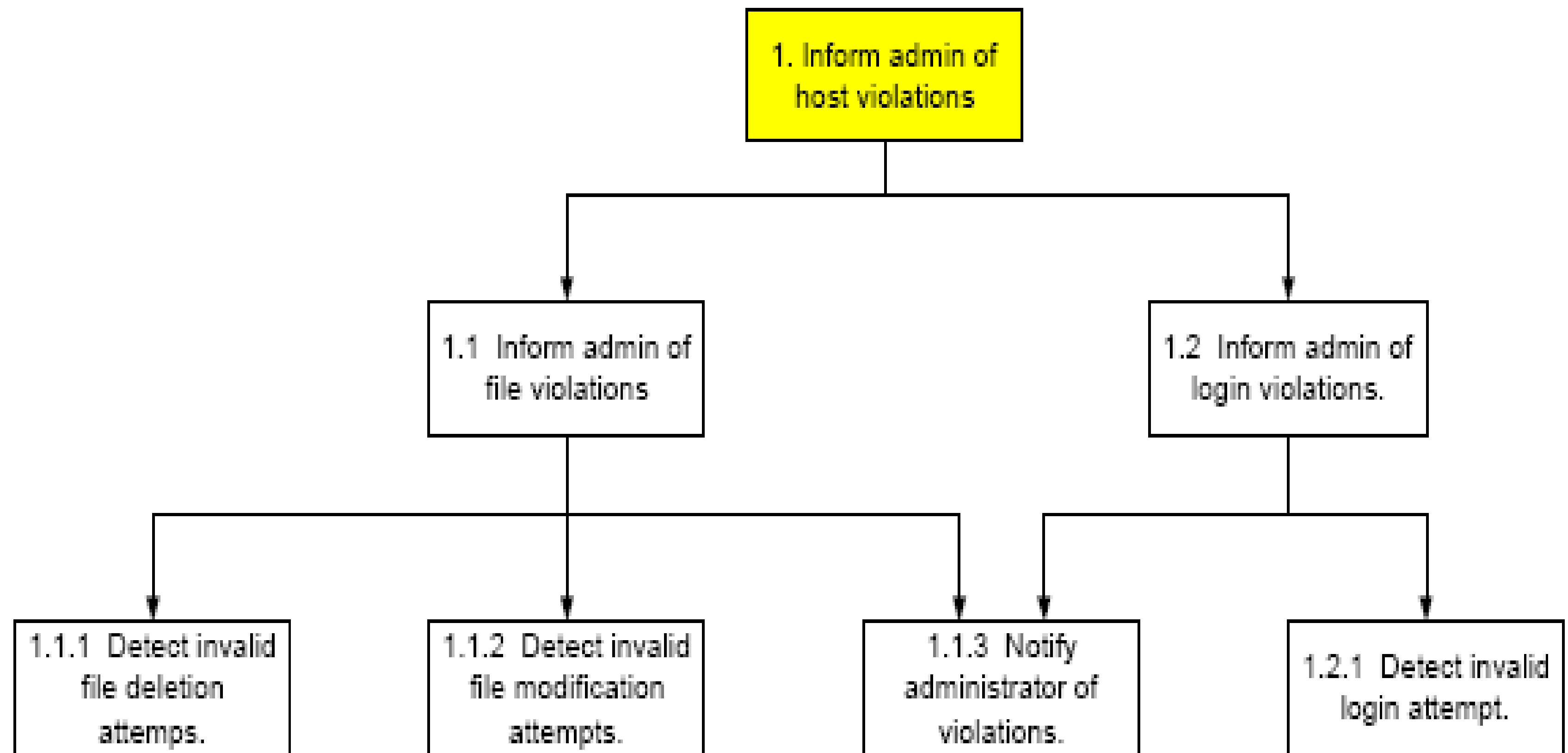
MaSE (*Analysis phase*)

There are two sub-steps in ***Capturing Goals***: identifying goals and structuring goals.

- First, goals must be identified from the initial system context. This process begins by extracting scenarios from the initial specification and describing the goal of that scenario.
- Next, the goals are analyzed and structured into a form that can be used later in the Analysis phase. In this stage the goals are structured into a Goal Hierarchy Diagram. A Goal Hierarchy Diagram is a directed, acyclic graph where the nodes represent goals and the arcs define a sub-goal relationship.

MaSE (*Analysis phase*)

Example of goal hierarchy



Applying Use Cases

The objective of the Applying Use Cases step is to capture a set of use cases from the initial system context and create a set of Sequence Diagrams to help the system analyst identify an initial set of roles and communications paths within the system. Use cases define basic scenarios that a system should be able to perform. The Sequence Diagrams capture the use cases as a set of events between the roles that make up the system. These event sequences are used later in the Analysis phase to define tasks that a particular role must accomplish.

MaSE (*Analysis phase*)

Refining Roles

The objective of the last step of the Analysis phase, Refining Roles, is to transform the structured goals and Sequence Diagrams into roles and their associated tasks, which are forms more suitable for designing multiagent systems.

Roles form the foundation for agent class definition and represent system goals during the Design phase.

It is our intention that system goals will be satisfied if every goal is associated with a role and every role is played by an agent class.

MaSE (*Design Phase*)

Design Phase

There are four steps to the designing a system with MaSE. The first step is ***Creating Agent Classes***, in which the designer assigns roles to specific agent types. In the second step, ***Constructing Conversations***, the actual conversations between agent classes are defined while in the third step, ***Assembling Agents Classes***, the internal architecture and reasoning processes of the agent classes are designed. Finally, in the last step, ***System Design***, the designer defines the actual number and location of agents in the deployed system.

MaSE

- Conclusion: MaSE is a comprehensive methodology for the analysis of multiagent systems and provides solid foundation for the design and development of multiagent systems. MaSE not only takes advantage of goaldriven development, but also uses the power of multiagent systems by defining roles, protocols and tasks in the analysis phase.

Comparing object-oriented methodologies.

- **A COMPARISON FRAMEWORK**

  The comparison framework covers four major aspects of each methodology: **Concepts, Modeling language, Process and Pragmatics.**

Comparing object-oriented methodologies.

- **Concepts:**

  Agent-oriented concepts are of great importance for agent-oriented methodologies in general and for agent-oriented modeling languages in particular. There a set of significant agent-oriented concepts was presented. These include the dentition of agents, their characteristics such as adaptability, mental notions (such as beliefs, desires and intention), the relationship and communication between agents, and other concepts.

Comparing object-oriented methodologies.

- **Modeling language:**

  If agent-oriented concepts are the basis for any AOSE methodology, then the modeling language for representing designs in terms of those concepts is generally the core component of any software engineering methodology. A typical modeling language consists of three main components: symbols (either graphical or textual representation of the concepts), syntax and semantics. It is important that the modeling language allows the system under development to be modeled from deferent views such as behavioral, functional and structural views .

Comparing object-oriented methodologies.

- The criteria which assess the modeling language of each methodology are categorized into two groups.
- **Usability criteria** reflects usage requirements of a modeling language in terms of providing a means for software developers to exchange their thoughts and ideas. These criteria basically addresses the question of how easy the notation and the models are to understand and to use.

Comparing object-oriented methodologies.

- The second group of criteria to assess a modeling language is technical criteria. They involve the unambiguity and consistency of a modeling language. Unambiguity means that a constructed model can be interpreted unambiguously. Consistency is a technical quality relating to the assistance of a modeling technique to the software designer in guaranteeing that between representations, no set of individual requirements is in conflict.

Comparing object-oriented methodologies.

- **Process:**

  As discussed above, the modeling language is considered as a mandatory part of any software engineering methodology. However, in constructing a software system, software engineering also emphasizes the series of activities and steps performed as part of the software life cycle. These activities and steps form the process which assists system analysts, developers and managers in developing software. An ideal methodology should cover enterprise modeling, domain analysis, requirements analysis, design, implementation and testing.

Comparing object-oriented methodologies.

- **Pragmatics:**

In addition to issues relating to notation and process, the choice of a methodology depends on the pragmatics of the methodology.

This can be assessed based on two aspects : management and technical issues.

Management criteria should consider the support that a methodology provides to management when adopting it. They include the cost involved in selecting the new methodology and its effects on the current organization business practices.

Technical criteria look at a methodology from another angle. They consider whether the methodology is targeted at a specific type of software domain such as information systems, real time systems or component-based systems.

Comparing object-oriented methodologies.

- Notation: L for Low, M for medium, H for High, DK for Don't Know, SDA for Strongly Disagree, DA for Disagree, NA for Not Applicable, N for Neutral, A for Agree, SA for Strongly Agree, for no response. S for Stage mentioned, P for Process given, E for Examples given, H for Heuristics given, n for none.

Concepts & Properties	**MaSE**	**Prometheus**
Autonomy	H/M/DK	H/NA/H
Mental attitudes	L/M/H	H/M/H
Teamwork	H/M/H	N/L/NA
Protocols	H	M/H/M
Agent-oriented	SA/A/A SA	SA
Modeling & Notation		
Syntax defined	A/A/SA	SA/A/A
Clear notation	A	SA/A/A
Easy to use	SA/A/A	A/N/A
Easy to learn	N/N/A	SA
Language adequate & expressive	SA/N/N	A
Traceability	A/SA/SA	A
Consistency check	SA/A/SA	SA/A/A
Modularity	SA/A/A	SA/SA/A
Hierarchical modeling	N/A/A	SA/A/A
Process		
Requirements	SPEH	SPEH
Architectural design	SPEH	SPEH
Detailed design	SPEH	SPEH
Implementation	SPEH	SPEH
Pragmatics		
Quality	N/DA/A	A/N/N
Management decision	/DA/SA	SDA/N/
Distributed	SA	SA

Comparing object-oriented methodologies.

- **Concepts:**

With regard to agent-oriented concepts, the level of support for autonomy of all of the methodologies is overall good (ranging from medium to high).

Prometheus supports very well the use of mental attitudes (such as beliefs, desires, intentions) in modeling agents' internals (medium to high), whereas MaSE provides weaker support.

The support for pro-activeness and reactiveness are difficult to measure even though they seem to be fairly well supported by all two methodologies (medium-high for MaSE and Prometheus).

In terms of support for concurrency, although the ratings are mostly medium-high and varied considerably, MaSE is probably best with its protocol analyzer, and Prometheus was rated as being one of the weakest.

Although the methodologies all support cooperating agents, none of them support teams of agents in the specific sense. Both MaSE and Prometheus model the dynamic aspects of the system and handle protocols well.

Comparing object-oriented methodologies.

- Modeling Language:

  Overall, the responders felt that the methodologies' notations were clear and reasonably well defined (syntax/semantics) and fairly easy to use.

  Modularity, and hierarchical modeling are generally well-supported , however reuse is not well handled by any of the methodologies.

  Very good impression of the notation of all the methodologies.

  Prometheus is highly appreciated, and the system overview diagram in particular was found to be useful. There are some cases where the amount of text on arcs in the MaSE's concurrent diagrams makes them hard to read.

Comparing object-oriented methodologies.

Process:

From the software development life-cycle point of view, all of the methodologies cover the requirements, architectural design and detailed design.

Analysis stage of the methodologies is well described and provides useful examples with heuristics. This helps to shift from object-oriented thinking to agent-oriented.

The implementation phase is, surprisingly, not well supported: MaSE and Prometheus mention testing/debugging, but it is unclear to what extent MaSE supports it, while Prometheus' support is part of a research project not yet integrated into tools for use by developers.

Comparing object-oriented methodologies.

- Pragmatics:

  The pragmatics of a methodology plays a very important role in determining its applicability in industry as well as in academia. MaSE and Prometheus target undergraduate and industry programmers. Regarding the availability of resources supporting the methodologies, most of them are in the form of conference papers, and journal papers or tutorial notes. None of the methodologies are published as text books.

Comparing object-oriented methodologies.

- **CONCLUSION**

  Overall, all two methodologies provide a reasonable support for basic agent-oriented concepts such as autonomy, mental attitudes, pro-activeness, reactiveness, etc.

  They all are also regarded by their developers and the students as clearly agent-oriented.

  In addition, the notation of the two methodologies is generally good.

  Regarding the process, all the methodologies provide examples and heuristics to assist developers from requirements gathering to detailed design.

  Implementation was supported to some degree by all methodologies whereas testing/debugging and maintenance are not clearly well-supported by any methodology.

  Additionally, some important software engineering issues such as quality assurance, estimating guidelines, and supporting management decisions are not supported by any of the methodologies.

The end