Agent-Oriented Software Engineering

Federico Bergenti
Onn Shehory
Franco Zambonelli

Goal of the Lecture

- Understand and discuss:
  - What Agent-Oriented Software Engineering (AOSE) is and why it is important;
  - Key concepts.
- Overview:
  - Relevant AOSE methodologies;
  - AOSE implementation tools.
- Suggest interesting research directions.

Outline of the Lecture

- Part 1 (≅2 hours): Key Concepts
  - Agents and multiagent systems from a software engineering perspective.
- Part 2 (≅2 hours): Methodologies
  - Overview and evaluation of 4 representative methodologies.
- Part 3 (≅1 hour): Implementation
  - Discuss approaches for implementing engineered agent-based systems, and their implications for methodologies.
- Part 4 (≅1 hour): The Road Ahead & Discussion
  - Open research directions and visions....

NOTE

- In this tutorial we present
  - 70% FACTS
  - 30% OPINIONS
- Please, feel free to contradict us!
  - We are not necessarily repositories of TRUTH!
  - Your ideas may be much fresher!

Part 1

Key concepts in agent-oriented software engineering

Part 1: Outline

- Why?
  - Software engineering, agents, agent-oriented software engineering (AOSE).
- The software engineering perspective on agent-based computing:
  - Pervasive, novel agent-oriented abstractions;
  - Engineering complex systems is engineering agent systems.
- New levels of abstraction in software development:
  - The knowledge level vs. the social level.
- Agent-based analysis and design:
  - At the knowledge level and at the social level.
What is Software Engineering

- Software is pervasive and critical:
  - It cannot be built without a disciplined, engineered, approach
- There is a need to model and engineer both:
  - The development process:
    • Controlable, well documented, and reproducible ways of producing software;
  - The software:
    • Well-defined quality level (e.g., % of errors and performances);
    • Enabling reuse and maintenance.
- Requires:
  - Abstractions, methodologies and tools.

Software Engineering Abstractions

- Software deals with “abstract” entities, having a real-world counterpart:
  • Numbers, dates, names, persons, documents ...
- In what term should we model them in software?
  • Data, functions, objects, agents ...
  • I.e., what are the ABSTRACTIONS that we have to use to model software?
- May depend on the available technologies!
  • Use OO abstractions for OO programming envs.;
  • Not necessarily use OO abstractions because they are better, even for COBOL programming envs.

Methodologies

- A methodology for software development:
  - Is intended to give discipline to software development.
  - Defines the abstractions to use to model software:
    • Data-oriented methodologies, object-oriented ones ...
    • Define the MINDSET of the methodology.
  - Disciplines the software process:
    • What to produce and when;
    • Which artifacts to produce.

The Classical “Cascade” Process

- The phases of software development:
  - Independent of programming paradigm;
  - Methodologies are typically organized around this classical process:
    • Inputs, outputs, internal activities of “phases”

Tools

- Notation tools:
  - To represent the outcomes of the software development phases:
    • Diagrams, equations, figures ...
- Formal models:
  - To prove properties of software prior to development
    • Lambda and pi calculus, UNITY, Petri-nets, Z ....
- CASE tools:
  - To facilitate activities:
    • Simulators, rapid prototyping, code generators.

Example: OO Software Engineering

- Abstractions:
  - Objects, classes, inheritance, services.
- Methodologies:
  - UDP (Rumbaugh), object-oriented analysis and design;
  - Centered around the object-oriented abstractions.
- Tools:
  - UML (standard), E-R, class lattices, finite state automata, visual languages ...
Why Agent-Oriented Software Engineering?

- Software engineering is necessary to discipline:
  - Software systems and software processes;
  - Any approach relies on a set of abstractions and on related methodologies and tools
- Agent-based computing:
  - Introduces novel abstractions
    - Requires clarifying the set of necessary abstractions
    - Requires adapting methodologies and producing new tools
- Novel, specific agent-oriented software engineering approaches are needed!

What are Agents?

- There has been some debate
  - On what an agent is, and what could be appropriately called an agent
- Two main viewpoints:
  - The (strong) Artificial Intelligence viewpoint:
    - An agent must be proactive, intelligent, and it must converse instead of doing client-server computing
  - The (weak) Software Engineering Viewpoint
    - An agent is a software component with internal (either reactive or proactive) threads of execution, and that can be engaged in complex and stateful interactions protocols

What are Multiagent Systems?

- Again….
  - The (strong) artificial intelligence viewpoint
    - A multiagent system is a society of individual (AI software agents) that interact by exchanging knowledge and by negotiating with each other to achieve either their own interest or some global goal
  - The (weak) software engineering viewpoint
    - A multiagent system is a software systems made up of multiple independent and encapsulated loci of control (i.e., the agents) interacting with each other in the context of a specific application viewpoint….

The Software Engineering Viewpoint on AO Computing

- We commit to it because:
  - It focuses on the characteristics of agents that have impact on software development
    - Concurrency, interaction, multiple loci of control
    - Intelligence can be seen as a peculiar form of control independence; conversations as a peculiar form of interaction
  - It is much more general:
    - Does not exclude the strong AI viewpoint
    - Several software systems, even if never conceived as agents-based one, can be indeed characterised in terms of weak multi-agent systems
- Let’s better characterize the SE perspective on agents…

Key Characteristics of Agents (SE Viewpoint)

- Basic
  - Autonomy & Proactivity
  - Situatedness
  - Interactivity
- Additional
  - Mobility & Locality
  - Openness
  - Learning & Adaptive Capabilities

Agent Autonomy

- Process-based and Object-based applications
  - global goal achieved via a global control scheme for the application entities
  - design by delegation of control
- Agent-based applications
  - sub-goals assigned to autonomous agents integrating execution capabilities, i.e., threads
  - implies perceiving agents as proactive entities
  - multiple independent loci of control in applications
  - design by delegation of responsibility
- SE Advantages
  - Control encapsulation as a dimension of modularity
  - Conceptually simpler to tackle than a single (or multiple inter-dependent) locus of control
Agent Situatedness

- Agents typically perceive a portion of the external world – an "environment"
  - Physical environment
    - A manufacturing plan, a room, etc...
  - Computational environment
    - A Web-site, an information system, etc...
- They have to sense and effect:
  - By perceiving what’s happening in the environment, and possibly influencing it:
    - control of manufacturing tools
    - access and update to Web data and services
- SE Advantages
  - Clear separation of concerns between:
    - the active computational parts of the system (the agents)
    - the resources of the environment

Agent Interactivity

- Agents may execute in multiagent contexts and interact with each other
  - Agent communication
  - Agent coordination
  - agents may be in need of orchestrating their activities
- Collaborative or competitive interactions
  - agents interact to achieve a common goal
  - competition as a peculiar form of collaboration
  - Useful goals achieved via self-interest (market models)
- SE implications
  - Not a single characterising protocol of interaction (e.g., client-server)
  - Interaction protocols as an additional SE dimension

Agent Mobility & Locality

- Autonomous components can migrate across different multi-agent systems (or contexts)
  - e.g., across different Internet nodes or domains
  - Interaction are limited to a context
- Non-Functional Motivations
  - save of bandwidth (local access to data and services)
  - robustness (independence from connection flaws)
  - intrinsic for software on mobile devices
- SE Motivations
  - Additional dimension of autonomous behavior
  - Improve locality in interactions
  - Reduce application complexity by sub-dividing systems into sub-systems and by identifying interaction localities

Openness of Multiagent Systems

- The agents in a system may not be fixed
  - New agents can be created or enter a multiagent systems context
  - Mobile agents can arrive
  - Unknown – legacy and elsewhere implemented – may enter a multiagent system
  - E.g., agent marketplaces must be by definition open
- Technological implications
  - Need of standards ! (e.g. FIPA)
  - Need of proper infrastructures supporting the interoperations
- SE Implications
  - Controlling self-interested agents, malicious behaviors, and badly programmed agents
  - Dynamic re-organization of software architecture

Learning and Adaptive Agents

- When agents have to be "intelligent"
  - They must be possibly able to learn from previous experiences
  - Improving the effectiveness of its actions
- When agents lives in dynamic scenarios
  - They must be able to adapt their behavior to changing situations
  - Re-shaping themselves
- SE is not concerned in
  - HOW learning and adaptiveness are achieved
- But it may be concerned with
  - WHAT could be the impact on the global software systems of having components that change their behaviour dynamically? (see open directions....)

MAS Characterisation
Agent-Oriented Abstractions

The development of a multiagent system should fruitfully exploit abstractions coherent with the above characterisation:

- **Agents**, autonomous entities, independent loci of control, situated in an environment, interacting with each others
- **Environment**, the world of resources agents perceive
- **Interaction protocols**, as the acts of interactions between agents

In addition, there may be the need of abstracting:

- The **local context** where an agent live (e.g., a sub-organization of agents) to handle mobility & openness

Such abstractions translates into concrete entities of the software system.

Agent-Oriented Methodologies

The is need for SE methodologies

- Centered around specific agent-oriented abstractions
- The adoption of OO methodologies would produce mismatches
  - Classes, objects, client-servers: little to do with agents!

Each methodology may introduce further abstractions:

- Around which to model software and to organize the software process
  - E.g., roles, organizations, responsibilities, belief, desire and intentions...
  - Not directly translating into concrete entities of the software system
  - E.g. the concept of role is an aspect of an agent, not an agent

Agent-Oriented Tools

SE requires tools to

- represent software
  - E.g., interaction diagrams, E-R diagrams, etc.
- verify properties
  - E.g., petri nets, formal notations, etc.

AOSE requires

- Specific agent-oriented tools
  - E.g., UML is not suitable to model agent systems and their interactions

Why Agents and Multiagent Systems?

Other lectures may have already outlined the advantages of (intelligent) agents and of multiagent systems, and their possible applications

- Autonomy for delegation (do work on our behalf)
- Monitor our environments
- More efficient interaction and resource management

Here, we mostly want show that

- Agent-based computing, and the abstractions it uses, represent a new and general-purpose software engineering paradigm!

There is much more to agent-oriented software engineering

AOSE is not only for "agent systems."

- Most of today’s software systems have characteristics that are very similar to those of agent and multiagent systems
- The agent abstractions, the methodologies, and the tools AOSE suit such software systems

AOSE is suitable for a wide class of scenarios and applications!

- Agents’ "artificial Intelligence" features may be important but are not central

Examples of components that can be modelled (and observed) in terms of agents:

- Autonomous network processes;
- Computing-based sensors;
- PDAs;
- Robots.

Example of software systems that can be modelled as multiagent systems:

- Internet applications;
- P2P systems;
- Sensor networks;
- Pervasive computing systems.

Agents and Multiagent Systems are Everywhere!
Internet Applications

- Components (Web processes) autonomous
  - Each component is a process in a site, independently developed and independently running, with an (observable) proactive behavior
- Components are situated
  - The data, services, and resources of a site, that the components can "sense" and effect
- Components are interactive
  - Interaction are based on client-server (at the lower level) too complex to tackle
  - Effective modeling requires modeling interactions from a higher-level perspective
- In addition
  - Components can be mobile (Java mobile code)
  - Interactions can be local (security)

P2P Systems

- Set of independent user-application
  - Representing autonomous loci of control
- Controlling a set of resources
  - E.g., files, mp3, CPU power, storage...
  - Therefore, situated in the environment represented by such resources
- Interacting with each other
  - To exchange data, files, CPU cycles, etc.
  - Exchanges not necessarily relying on client-server interactions 
  - negotiations for use of resources
- In addition:
  - P2P communities intrinsically open and dynamic
  - Spontaneous (re-)organization

Sensor Networks

- Micro-computer-based systems
  - Autonomous by definition
- Situated in an environments
  - Dispersed outdoor (or in manufacts) and devoted to monitor that a portion of the environment
- Interactive (wireless or optical connections)
  - Global monitoring data must be organized from local observation
  - Such data organization may require complex – non client-server – interactions and negotiations
- In addition
  - Local interactions (limited comms. capability)
  - Openness: sensors die and new can be added

Pervasive Computing Applications

- Characterization similar to sensor networks:
  - Autonomy: computers and devices dispersed everywhere
  - Situated: devoted to monitor and effect specific portion of the environment
  - Interactive: to provide services to anyone, and to coordinate composite services
- In addition:
  - Mobile: we can wear computers, we can move computer based objects around the world (e.g., a TV in different rooms, a car in different nations)

Summarizing

- A software engineering paradigm define:
  - The mindset, the set of abstractions to be used in software development and, consequently,
  - Methodologies and tools
  - The range of applicability
- Agent-oriented software engineering defines
  - Abstractions of agents, environment, interaction protocols, context
  - Of course, also specific methodologies and tools (in the following of the tutorial)
  - Appears to be applicable to a very wide range of distributed computing applications...

Getting Deeply into Agent-Oriented Software Engineering…

- For the definition of a suitable methodology for multiagent systems development (and for presenting methodologies in this lecture…)
  - there is need of better characterizing agents, multi-agent systems, and the associated mindset of abstractions
    - How can we model agent autonomy, situatedness and sociality
  - There is need of understanding how the “traditional” cascade software engineering process maps into agent-oriented software development
    - What are analysis and design in AOSE?
Characterizing Agents

- No agreement on the definition of agent.
- Historically, two approaches to characterize "intelligent", i.e., rational, agents and multiagent systems:
  - Operational: agents and multiagent systems are systems with particular features, i.e.,
    - Particular structure;
    - Particular behaviour;
  - Based on system levels: agents and multiagent systems are new system levels.
- These approaches are complementary.

Operational Characterization

- Particularly suited for rational agents because it is based on logics.
- Rational agents (Wooldridge):
  - Described in terms of a belief, desires and intention;
  - Beliefs, desires and intentions are structured so to make the agent behave rationally;
  - Independent from the internal agent architecture;
  - The whole work on LORA devoted to it.

Operational Characterization

- Simple control loop of a rational agent:
  1. forever
  2. sense the environment
  3. update the model of the environment
  4. deliberate for a new goal
  5. means-end reason to obtain a plan to achieve the goal
  6. execute the plan

Operational Characterization

- The operational characterization:
  - Draws from well-founded logics;
  - Does not depend on the internal architecture of the agents.
- This approach has, at least, two problems:
  - Does not justify reasonably why we should adopt agents instead of other technologies;
  - Grounds rationality on the axioms of a logic;
  - Could not make any accepted agreement.

System Levels

- System level: structured group of concepts that support the definition of an engineered model of a system.
- Historically, introduced to hide details in hardware design, e.g.:
  - A logic gate level design does not care about transistors;
  - A register transfer level design does not care about gates.
- System levels are levels of abstraction.
Example: Logic Gate Level

<table>
<thead>
<tr>
<th>Element</th>
<th>Logic Gate Level Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Unit of a processor that manipulates registers</td>
</tr>
<tr>
<td>Components</td>
<td>Logic gates, lines</td>
</tr>
<tr>
<td>Medium</td>
<td>Single-bit signals</td>
</tr>
<tr>
<td>Composition</td>
<td>E.g., input and output of logic gates are connected through lines</td>
</tr>
<tr>
<td>Behaviour Law</td>
<td>The laws for composing truth tables of logic gates</td>
</tr>
</tbody>
</table>

Knowledge Level

- At the beginning of the 80's the AI had the problem of defining knowledge.
- Introduced a new system level, called knowledge level, to provide a scientific definition of knowledge (Newell).
- The knowledge level is used to model agents, i.e., rational systems that process knowledge.

Knowledge Level

- The knowledge level:
  - Relies only on the principle of rationality to account for the behaviour of the agent;
  - Focuses on modelling one single agent.
- Today, we build systems in terms of:
  - Agents that may not be proved to be rational at all;
  - Interacting agents that are the unit of reuse and of encapsulation.

Social Level

- Jennings introduced the social level on top of the knowledge level.
- It allows to create organizational models of multiagent systems.

Social Level

- The social level:
  - Moves all design towards social issues, does not care of how to design each agent;
  - Cannot describe emerging organizations.
- Best practice of architectural patterns suggests that organization is not enough to design a system, e.g., we need:
  - Connectors for flexible composability;
  - Contracts to support verifiable composability.
Agent Level

<table>
<thead>
<tr>
<th>Element</th>
<th>Agent Level Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Multiagent system</td>
</tr>
<tr>
<td>Components</td>
<td>Belief, goal, action, role, interaction rule</td>
</tr>
<tr>
<td>Medium</td>
<td>Representation of belief, goal and capabilities</td>
</tr>
<tr>
<td>Behaviour Law</td>
<td>Principle of rationality</td>
</tr>
</tbody>
</table>

- Between the knowledge and the social level.
- Allows to model multiagent systems that:
  - Rely on message passing and on the speech-act theory;
  - Exploits the possibilities of the FIPA infrastructure.

Agents and Other Technologies

- Since FIPA, multiagent systems are often compared with object-oriented systems:
  - Both rely on encapsulated units that interact;
  - Both rely on message passing;
  - For both we can define an architecture;
  - … and many other similarities.

- The comparisons found in the literature are often poor.

“Agents can say…”

- Use of autonomy to draw a line between agents and objects (Parunak):
  - “Agents can say go,” i.e., agents can take the initiative;
  - “Agents can say no,” i.e., agents can refuse to perform a requested service.
- These seem relevant differences with object-oriented method invocation, but:
  - Active objects have a long and honored history;
  - Refusal is not useful per se.

Comparing the Meta-Models

<table>
<thead>
<tr>
<th>Element</th>
<th>Objects</th>
<th>Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Properties and values</td>
<td>Knowledge base</td>
</tr>
<tr>
<td>Messaging</td>
<td>Request for service with certain parameters</td>
<td>Exchange of parts of the knowledge base</td>
</tr>
<tr>
<td>Reuse</td>
<td>Inheritance, mostly for composability</td>
<td>Composability</td>
</tr>
<tr>
<td>Delegation</td>
<td>Task delegation</td>
<td>Goal delegation</td>
</tr>
<tr>
<td>Responsibility</td>
<td>Design by contract</td>
<td>Pre/post conditions</td>
</tr>
<tr>
<td>Type system</td>
<td>Classes</td>
<td>None</td>
</tr>
</tbody>
</table>

Comparing Granularity

- Objects have a highly dynamic lifecycle, they are:
  - Created just for serving a request;
  - Cloned just for performance reasons;
  - Introduced to promote reusability;
  - … often created and destroyed.
- Agents are more coarse grained:
  - Reason on their knowledge bases;
  - Publish their capabilities to a DF;
  - … they are rarely created and destroyed.

Agents and Components

- It seems more reasonable to compare agents against software components, e.g., EJBs, CORBABeans and .NET Components.
- They have the same granularity and FIPA provides a similar infrastructure.

- The questions still remain:
  - What are the advantages of using an agent instead of a component?
  - When shall we chose one or the other?
Agent-Based Analysis

Analysis aims to understand, at least:
- What are the main actors interacting with the system;
- How the system interacts with these actors;
- What the system is supposed to do.

The system is a closed entity and we do not look into it to avoid anticipating design issues and decisions.

Where do agents enter the picture?

We associate agents with the entities of the scenarios we are analyzing.

Then, we associate accordingly:
- Roles, responsibilities and capabilities;
- Interaction patterns between agents.

This provides a neutral view of the problem.

Methodologies, e.g., Tropos and GAIA, do not use the word agent to identify analysis-phase entities.

Analysis at the Knowledge Level

We need to identify for each agent:
- Its beliefs;
- Its goals;
- Its body, i.e., the way it interacts with the environment;
- Its actions on the environment.

We need to identify the behavior of the environment, i.e., how it interacts with the body and reacts to actions.

Interactions with other agents are mediated through the shared environment:
- No explicit communication.

Avoiding the identification of any architecture accounts for self-organization.

Various methodologies provide different:
- Artifacts;
- Paths in the process;
To identify these elements.

The elements are an alternative view, and can be derived from:
- The responsibilities of the agent;
- The capabilities of the agent;
- Any implicit organization within the agent system.

Focus on the analysis of an organization made of agents.

We need to identify:
- The roles in the organization;
- The organizational relations between roles;
- The dependency between roles;
- The interaction channels;
- The obligations;
- The influence mechanisms.
Analysis at the Social Level

- Interactions between agents is mediated through the possible interactions between the roles:
  - No explicit communication.
- The identification of a fixed architecture prohibits self-organization.

Agent-Based Design

- Design aims to engineer, at least:
  - What are the main components interacting within the system;
  - What are the responsibilities and the capabilities of each component in the system;
  - How the components interact to implement the system, i.e., the architecture of the system.
- Where do agents enter the picture?

Agent-Based Design

- We associate agents with the components we use to build the system.
- Then, we associate accordingly:
  - Roles, responsibilities and capabilities;
  - Interaction patterns between agents.
- Differently from analysis: we need to choose on which agents to use and how they interact.
- Agents at the design phase can have nothing to do with agents at the analysis phase.

Design at the Knowledge Level

- We need to decide for each agent:
  - Its beliefs;
  - Its goals;
  - Its body, i.e., the way it interacts with the environment;
  - Its actions on the environment.
- We also need to decide the behavior of the environment, i.e., how it interacts with the body and reacts to actions.

Design at the Knowledge Level

- The design decisions are on the basis of consideration regarding:
  - Reusability;
  - Performances;
  - Maintainability;
  - … and all other features we would like our software to have.
- No design patterns have been identified for knowledge-level design.

Design at the Social Level

- We need to choose for our society, i.e., for our architecture:
  - The roles in the organization;
  - The organizational relations between roles;
  - The dependency between roles;
  - The interaction channels;
  - The obligations;
  - The influence mechanisms.
- We do not consider the environment at the social level.
Design at the Social Level

- The design decisions are on the basis of consideration regarding:
  - Reusability;
  - Performances;
  - Maintainability;
  - ... and all other features we would like our software to have.
- We have design patterns for this kind of design in organizational theory.

Part 2
Overview and evaluation of agent-oriented software engineering methodologies

Outline

- What is a methodology (reminder)?
- How can we evaluate it?
  - Evaluation techniques
  - Criteria for featured based evaluation
- Existing Agent-Oriented Methodologies
- Methodologies Overview and Evaluation
  - GAIA
  - AUM (Agent Unified Modeling Language)
  - DESIRE (DEsign and Specification of Interacting Reasoning)
  - OPM/MAS (Object-Process Methodology for Multi-Agent System)

What is a methodology?

To evaluate a methodology, we need to recall what a methodology is:

1: a body of methods, rules, and postulates employed by a discipline: a particular procedure or set of procedures
2: the analysis of the principles or procedures of inquiry in a particular field

(Merriam-Webster)

Evaluation techniques

- Feature comparison – a comparison that follows a set of ideal modeling technique features
  - Advantages: Easy to perform if criteria are well defined
  - Drawbacks: subjective
- Meta-modeling – a comparison of meta-level of modeling techniques by mapping them to a super modeling technique or comparing their parts
  - Advantages: more objective
  - Drawbacks: not sufficiently objective
- Metrics – a comparison of the formal meta-modeling according to pre-defined metrics (such as number of constructs)
  - Advantages: objective
  - Drawbacks: a lot of empirical work is needed
Evaluation techniques

- Ontological evaluation – a comparison of exiting vs. needed constructs (e.g., system, event)
  - Advantages: has a strong theoretical foundation
  - Drawbacks: difficult to justify foundation choice
- Survey – gathers data on attitudes, opinions, impressions and beliefs of human subjects
  - Advantages: enables gathering information regarding the ways subjects understand/perceive the technique
  - Drawbacks: low response rate to questionnaires, the results characteristics are subjective

Evaluation techniques

- Laboratory experiment – enables manipulation of independent variables (e.g., modeling technique) and measurement of the effect on the dependent variable (e.g., accuracy)
  - Advantages: enables control over variables
  - Drawbacks: may not reflect the real world
- Field experiment – same as laboratory experiment but performed within organizations
  - Advantages: real-world experiment
  - Drawbacks: difficult to conduct
- Case study – there is no intervention of the evaluator
  - Advantages: promotes acceptance by organizations
  - Drawbacks: subjective

Feature-based evaluation

- We compare features of several methodologies
- For this, we initially list the features to be compared, classified as follows:
  - Agent-based system characteristics
  - Software engineering criteria
- The properties selected are a subset of the available, however they are perceived by many in the community as the important properties of agent-based systems.
- By comparison, we learn strengths and weaknesses of methodologies and differences between them

Agent-Based System Characteristics

- Autonomy: unlike objects, agents may be active and are responsible for their own activities. An agent has control over both its reactive and proactive behaviors
- Complexity: agent-based systems are basically sets of components (agents) that interact with each other in order to achieve their goals. These systems may consist of decision-making mechanisms, learning mechanisms, reasoning mechanisms, and other complex algorithms
- Adaptability: agent-based systems have to be flexible in order to adjust their activities to the dynamic environmental changes

Agent-Based System Characteristics

- Concurrency: an agent may need to perform several activities or tasks at the same time. The concurrency requirement implies that an agent-based system must be designed to carry out parallel processing
- Communication richness: a definition of an agent includes its autonomous activity. As an autonomous entity, an agent must establish communication with its environment, which may include other agents and information sources. The communication is characterized by its type (inter-agent or intra-agent) and its content
- Distribution: Multi-agent systems often operate on different hosts and are distributed over a network
- Mobility: An agent might sometimes want to transport itself from one environment or platform to another
- Security and privacy: Due to agents’ social activities, they might be exposed to intrusion to their data, state, or activities. Agents might want to keep some information for themselves or reveal it just to a specific entity (e.g., another agent)
- Openness: Multi-agent systems are sometimes flexible in the sense they can dynamically decide upon their participants
Software Engineering Criteria

- **Preciseness**: the semantics of a modeling technique must be unambiguous in order to avoid misinterpretation of the developed models by those who use it.
- **Accessibility**: a modeling technique should be comprehensible to both experts and novices.
- **Expressiveness**: a modeling technique should be applicable across multiple domains and represent the following aspects: the structure of the system; the knowledge encapsulated within the system; the system ontology and relationships with other system aspects; the data flow within the system; the control flow within the system; the resource constraints (i.e., time, CPU and memory); the system’s physical architecture.

- **Modularity**: a modeling technique should be expressible in stages. That is, when new specification requirements are added, there is no need to modify pervious parts, and these may be used as part of the new specification.
- **Complexity management**: a modeling technique should be expressed, and then examined, at various levels of detail.
- **Executability**: a prototyping capacity or a simulation capacity should be associated with at least some aspects of the modeling technique.

Let’s Now Overview & Evaluate…

- **Existing Agent-Oriented Methodologies**
  - GAIA
  - AUML (Agent Unified Modeling Language)
  - DESIRE (DEsign and Specification of Interacting Reasoning)
  - OPM/MAS (Object-Process Methodology for Multi-Agent System)

Agent-Oriented Methodologies - Knowledge Engineering Approach

Knowledge engineering is the process of eliciting, structuring, formalizing and operationalizing information and knowledge.

- **Advantages**
  - Provides techniques for modeling the agent’s knowledge
  - Does not address software engineering criteria

- **Drawbacks**
  - Agent communication is not just method invocation
  - There is no reference to the mental state of the agent

- **Examples**
  - DESIRE (Treur, Jöker, Brauer)
  - MAS-CommonKADS (Iglesias, Garijo, Gonzalez, Velasco)
  - …

Agent-Oriented Methodologies - Software Engineering Approach

Software Engineering (OO) is the application of a systematic, disciplined, quantifiable approach (OO) to development, operation, and maintenance of software.

- **Advantages**
  - Some claim that an agent is an active object (thus OO provides everything we need)
  - Commonly used and popular

- **Drawbacks**
  - Agent communication is not just method invocation

- **Examples**
  - AUML (Agent Unified Modeling Language) – Odell and al.
  - DESIRE – Jennings, Faratin, Norman, O’Brien
  - MESSAGE/UML – EURESCOM Project
  - OPM/MAS – Sturm, Dori, shehory
  - …
Evaluating Methodology

- **GAIA**
  - Represents an extension of the software engineering approach
  - Has a solid social foundation
- **AUML**
  - Represents the pure software engineering approach
  - It is an extension of the standard software engineering approach - UML
- **DESIRE**
  - Represents the pure knowledge engineering approach
  - Has proven capabilities (prototypes)
- **OPM/MAS**
  - Represents an approach combining object-orientation and process-orientation
  - Has been evaluated systematically

Evaluation Process

- For each methodology:
  - The basic models and guidelines are presented
    - We do not present the methodology, rather, we present its modeling technique (i.e., the analysis and design stages)
  - A case study demonstrates the modeling technique and is used to compare the capabilities of different techniques

Case Study

**Auction agent**

1. The *configurator*: a GUI component, enables the user to control and monitor the agent’s activity
2. The *parser*: translates retrieved information into an internal structure
3. The *bidder*: submits bids according to buying strategy. Implements two stages, bid and confirmation
4. The *manager*: controls the agent’s activity, monitors the auction site, activates the parser, determines the next bid, activates the bidder and terminates the agent’s purchasing activity

GAIA – Methodology Map

- The analysis phase consists of the following models:
  - Role definition (permissions, responsibilities and protocol)
  - Interaction model (used for protocol description)
- The design phase consists of the following models:
  - Agent model
  - Service model (input, output, pre and post condition)
  - Acquaintance model

GAIA – Role model

- The *permissions* attribute states what resources may be used to carry out the role and what resource constraints the role’s executor is subject to
- The *responsibilities* attribute determines the functionality of the role. This functionality is expressed in terms of safety and liveness properties
- The *protocol* attribute states the interactions of the role with other roles. In addition it states the internal activities of the role
GAIA – Role model

**Role Schema: Manager (MA)**

- **Description:** Controls the auction agent activities
- **Protocols and Activities:**
  - CheckAuctionSite
  - ActivateParser
  - CheckForBid
  - Bid
- **Resources:**
  - ItemNumber
  - AuctionDetails
- **Responsibilities:**
  - Liveness: Manager = (CheckAuctionSite . ActivateParser . CheckForBid ) + Bid
  - Safety: true

The Manager role scheme

---

GAIA – Interaction Model

**Interaction Model of the CheckAuctionSite protocol**

- **Sender:** Manager
- **Receiver:** AuctionSite
- **Description:**
  - Connect to the auction site for auction status and information

---

GAIA – Design Phase Models

**The Agent Model**

- **Service Model**
  - **Services:**
    - Get auction details: ItemNumber → AuctionDetails
    - Validate user: User → Exists (exists=true) ∨ (success=false)
    - Bid: User, ItemNumber, Price → Success (user exists) ∨ (success=false)

**The Service Model**

---

GAIA – ABC Evaluation

- **Autonomy:** the role encapsulates its environment
- **Adaptability:** optional execution can be expressed by the liveness properties
- **Openness:** a generic agent can be modeled and may represent any new agent that will participate within the MAS application
- **Complexity:** difficult to model complex computation
- **Concurrency:** this issue is not dealt with
- **Communication richness:** message content and architecture are not dealt with
- **Distribution:** this issue is not dealt with
- **Mobility:** this issue is not dealt with
- **Security and privacy:** this issue is not dealt with explicitly

---

GAIA – SE Evaluation

- **Preciseness:** provided via the liveness and safety properties
- **Accessibility:** simple and clear models
- **Expressiveness:** generic structure, can handle various systems. Flatness may restrict to small-medium systems. No explicit representation of data/control flow, knowledge, structure
- **Modularity:** use of building blocks supports modularity, but changes within a role may cause a chain reaction of changes
- **Portability:** no limitations
- **Complexity management:** no hierarchical presentation or another mechanism for complexity management
- **Executability:** in GAIA this issue is out of scope
- **Refinability:** GAIA provides guidelines on moving from analysis to design, but no guidelines for moving toward implementation
- **Analyzability:** no CASE tool is provided

---

AUML

- **Agent Unified Modeling Language is based on UML**
- **AUML is not a language yet, it is a proposal**
- **Extended with the following:**
  - Organized special agent class
  - New concept of role
  - New Agent Interaction Protocol Diagrams
AUML – Methodology Map

<table>
<thead>
<tr>
<th>Diagram Type</th>
<th>Class</th>
<th>Sequence</th>
<th>Use Case</th>
<th>Collaboration</th>
<th>Deployment</th>
<th>Subsystem</th>
<th>Package</th>
<th>Activity</th>
<th>Statechart</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>+</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>+</td>
<td>+</td>
<td></td>
<td>-</td>
<td></td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use Case</td>
<td>-</td>
<td></td>
<td></td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collaboration</td>
<td>+</td>
<td>+</td>
<td></td>
<td>-</td>
<td></td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deployment</td>
<td>-</td>
<td></td>
<td></td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subsystem</td>
<td>-</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
<td>-</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Package</td>
<td>-</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>+</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
<td>-</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statechart</td>
<td>-</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td>-</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AUML – Agent Notions

- **agent-class-name**: The class name of the agent.
- **role-name1, role-name2, …**: The names of the roles played by the agent.
- **state-description**: The description of the agent's state.
- **actions**: The actions that the agent can perform.
- **methods**: The methods that the agent can invoke.
- **methods-a()**: A method of the agent).
- **methods-m()**: Another method of the agent.
- **state-description**: The description of the state.
- **fields**: The fields of the agent.
- **agent-head-automata-name**: The name of the agent's head automata.
- **constraint**: A constraint that applies to the agent.
- **society-name**: The name of the society to which the agent belongs.
- **capabilities, service descriptions, supported protocols**: The capabilities, service descriptions, and supported protocols of the agent.
- **not-understood**: A protocol that the agent does not understand.
- **default**: The default protocol used by the agent.
AUML – Agent Interaction Protocol

- Layered protocol
- Nested protocol
- Interleaved protocol

Extending the behavioral diagrams to be fitted to the Role concept.

AUML – Knowledge Structure

- Class Diagram

AUML – System Structure

- Class Diagram

AUML – Agent Interaction Protocol

The protocol diagram of: Retrieve Information from the Auction Site

AUML – ABC Evaluation

- Autonomy: can be expressed within the agent class
- Adaptability: agent flexibility is modeled within the behavioral diagrams
- Concurrency: can be expressed using the sequence and protocol diagrams
- Communication richness: a good definition of communication through the protocol diagrams
- Distribution: can be expressed using the deployment diagram

Openness: in AUML, an agent can be generic and thus represents unfamiliar agents.

Mobility: there are some extensions for UML to support agent mobility (not in AUML), but they do not support the dynamic nature of mobility.

Security and privacy: use case diagrams can be used as an authentication and access control mechanisms.

Complexity: modeling complex algorithms using AUML might be exhausting.
**AUML – SE Evaluation**

- **Preciseness**: AUML is not a language yet, there are no formal definitions.
- **Modularity**: supported by the OO paradigm.
- **Complexity Management**: supported via packages, models and subsystems.
- **Executability**: AUML as a descendant of UML can use the techniques of UML for rapid prototyping. It can be code skeleton or working application through statecharts (Rhapsody of I-Logix).
- **Refineability**: AUML as a descendant of UML can use the UML guidelines for refinements and the Rational Unified Process (RUP) for system development.
- **Analyzability**: AUML can be used within the existing tools of UML, thus it can take advantage of their capabilities.
- **Portability**: not coupled to a specific language or architecture.
- **Accessibility**: integrating AUML models difficult to understand and implement.
- **Expressiveness**: AUML is lacking in depicting knowledge representation and logical reasoning. In addition, it is lacking in depicting some dynamic aspects of the system.

**DESIRE**

- **DESIGN and Specification of Interacting REASONING framework**
- **DESIRE supports the following aspects**
  - Knowledge structure
  - Task composition
  - Information exchange
  - Task sequencing
  - Task delegation

**DESIRE – Methodology Map**

- **The Problem Description** includes the requirements imposed on the design
- **The Design Rational** specifies the choices made during the design process
- **The Conceptual Design** includes conceptual model for each agent, for the external world and agents' interactions
- **The Detailed Design** specifies all aspects of a system knowledge and behavior
- **The Operational Design** specifies the parameters needed for the implementation

We will focus on the conceptual and detailed design which consist of knowledge composition and process composition.

**Knowledge Representation - General**

- **sort** – can be viewed as a representation of a part of the domain
- **object** – is an instance of sort
- **function** – maps between sets of sorts
- **relation** – is a concept needed to make a statement
- **meta description** – a mechanism that enables association of world state to a specific sort
- **information type** – is a specification of a set of sorts, relations, objects and functions

**Knowledge Graphical Representation - Example**

- **domain**
- **domain price**
- **domain product**
- **domain action**
- **_price info**
- **action info**
- **domain info**
- **product info**
- **price info**
- **price has price**
- **price info**
- **price of product**
- **price of product**
- **price info**
- **price of product**
- **price info**
- **price of product**
Knowledge Textual Representation - Example

information type domain_info
  information types
    domain_prices, domain_products, prices_info;
  end information type
information type domain_prices
  sorts PRICE;
  objects $100, $200, $300 : PRICE;
end information type
information type domain_products
  sorts PRODUCT;
  objects A, B, C : PRODUCT;
end information type

Knowledge Textual Representation - Example

information type price_info
  sorts PRODUCT, PRICE;
  relations has_price: PRICE*PRODUCT;
end information type
information type domain_actions
  sorts ACTION;
  objects Bid, Stop, Wait: ACTION;
end information type
information type products_info
  sorts PRODUCT;
  functions comparison: PRODUCT * PRODUCT -> PRODUCT;
  relations is_cheaper: PRODUCT
end information type

Knowledge Base

knowledge base auction_kbs
  information types domain_info;
  contents
  if is_cheaper(p: PRODUCT)
    then perform_action (p: PRODUCT, Bid);
end knowledge base

DESIRE – Generic Agent Model

The task control in DESIRE has an equivalent textual representation
DESIRE – ABC Evaluation

- Autonomy: can be expressed within the task control knowledge
- Adaptability: agent flexibility is modeled within the task control knowledge
- Complexity: DESIRE provides tools for modeling complex algorithms
- Concurrency: this issue is not explicitly addressed
- Communication richness: no support for ACL, the depicted communication is between components
- Distribution: this issue is not dealt with
- Mobility: this issue is not dealt with
- Security and privacy: this issue is not dealt with
- Openness: in DESIRE all components must be defined explicitly, thus it cannot be configured dynamically

DESIRE – SE Evaluation

- Preciseness: provided via temporal logic
- Accessibility: DESIRE has a wide range of modeling capabilities which result in difficulty to learn and implement it
- Expressiveness: DESIRE is lacking in depicting computational algorithms
- Modularity: supported via the component model
- Complexity Management: hierarchical presentation supported within task hierarchy and components
- Executability: DESIRE has prototype generation capabilities
- Refinability: DESIRE is not assigned to a specific development stage, however enables refinement at any stage
- Analyzability: correctness and coverage are checked using formal specification
- Portability: at this stage DESIRE is coupled to its own architecture and implementation

OPM/MAS – OPM General

- Object-Process Methodology (OPM) is an integrated approach to the study and development of systems in general and information systems in particular.
- OPM unifies the system’s structure and behavior throughout the analysis, design and implementation of the system within one frame of reference using a single diagramming tool - the Object-Process Diagram (OPD) and a corresponding, English-like language - the Object-Process Language (OPL).

OPM Concepts

- Object
- Process
- Characterization
- Generalization
- Aggregation
- Invocation
- Exception
- Relation

Structural Relationships

- Event
- Condition
- Instrument
- Effect
- Result/Consumption
- Invocation
- Exception
A Platform is a machine that hosts a program or a system.

An Agent is a software-based computer system with autonomy, social ability, reactivity, and pro-activeness properties.

A Role is a socially expected behavior pattern usually determined by an individual's status in a particular society.

A Task is an assigned work often to be finished within a certain time.

An Object is an information entity.

An Ontology Term is information, which represents terms that might be synonyms to a specific object or task.

An Interaction is a message. The message could be one of the following types: Web type, Agent type, and System type.

Messaging is a process for handling the construction and understanding of interactions.

The domain level is a visual constraints language.

It is defined via a set of OPDs that represents the characteristics of the domain building blocks and their relationships.

The OPDs used are either unfolding (for characterization) or zooming-in (for detailed constraints).
Zooming into the Trader role

Zooming into the Searching task

Zooming into the Decision Making task

Zooming into the Bidding task

Unfolding of the System Information Objects

OPM/MAS – ABC Evaluation

**Unfolding of the System Information Objects**

**OPM/MAS – ABC Evaluation**

- **Autonomy**: In OPM/MAS, the autonomy aspect is implemented by encapsulating the agent activities within roles and the agent itself.
- **Adaptability**: OPM/MAS addresses the adaptability aspect via an event mechanism.
- **Concurrency**: The concurrency aspect is addressed in OPM/MAS via the invocation concept and graphical layout.
- **Communication richness**: In OPM/MAS, the communication richness is addressed using the Platform element.
- **Distribution**: OPM/MAS supports the distribution aspect using the Platform element.
- **Openness**: OPM/MAS does not require explicit specification of all MAS participants. Generic modeling of unknown agents will suffice.
- **Mobility**: The mobility aspect is not modeled explicitly, but it can be modeled using the core OPM expressive power.
- **Security and privacy**: The security aspect is not modeled explicitly, however, it can be modeled using core OPM expressive power.
- **Complexity**: OPM/MAS supports computational complexity but not logic reasoning complexity.
OPM/MAS – SE Evaluation

- **Precision**: OPM has clear semantics
- **Accessibility**: OPM is highly accessible due to its single model approach and scaling mechanisms (examined)
- **Modularity**: OPM supports modularity of both objects and processes
- **Complexity Management**: Complexity management mechanisms are integrated into the OPM model
- **Refinability**: OPM provides for refining of things within the OPM model and a set of rules for converting the diagrams into an executable code
- **Analyzability**: OPM is supported by CASE tools
- **Portability**: OPM is a generic methodology, and is not coupled with any programming language or architecture
- **Executability**: OPM seems to have the capability of generating a complete running application although this has not yet been fully implemented and tested
- **Expressiveness**: OPM is highly expressive, however it is lacking in human interface and knowledge representation aspects

Summary

- Multiple methodologies exist
- Agent-based system characteristics are well supported
- Software engineering properties are supported by some, to a limited extent
- It is necessary to select a specific methodology according to its suitability to the domain and function of the intended MAS application
  - GAIA – social-oriented applications small-medium scale
  - AUM – computational-based applications (e-commerce)
  - DESIRE – knowledge-based applications
  - OPM/MAS – computational-based applications (e-commerce)

Part 3: Outline

- Influence of implementation technology on Analysis and Design
- Implementation issues:
  - Implementing agents:
    - Internal agent architecture;
    - Communication architecture;
    - Implementation tools.
  - Implementing multiagent systems:
    - Communication infrastructures;
    - Coordination infrastructures;
    - Institutions.

Issues in Implementing Agents and Multiagent Systems

- How can we move from agent-based design to concrete agent code?
- Methodologies should abstract from:
  - Internal agent architecture;
  - Communication architecture;
  - Implementation tools.
- However, depending on tools the effort from design to implementation changes:
  - It depends on how much abstractions are close to the abstractions of agent-oriented design.

Implementing Agents

- We have two categories of tools to implement agents:
  - Object-oriented tools: are very much related to the object-oriented approach, e.g., frameworks;
  - BDI toolkits: are based on the BDI model.
- The choice of the tool to adopt is hard and there is no general answer:
  - Performances;
  - Maintenance;
  - ... and many other issues.
Object-Oriented Tools: JADE

- JADE (Java Agent DEvelopment framework) implements a FIPA platform. It:
  - Is distributed across the network in terms of containers;
  - Provides management facilities, e.g., RMA;
  - Supports concurrent development facilities, e.g., JADE.
- The agent architecture is based on frameworks that implement the tasks of the agent:
  - One agent runs in one thread;
  - Cooperative scheduling of prioritized behaviors.
- Different type of behaviors, e.g.:
  - Plan;
  - Cycle;
  - FSM;
  - Cooperative scheduling of prioritized behaviors.

ParADE (Parma Agent Development Environment) is a toolkit for the development of BDI FIPA agents.

Agent level:
- Agents are atomic components;
- UML is used to build models of single agents and of the multiagent system.

Object level, exploits the generated code:
- Each agent is an object-oriented system;
- ParADE provides is a framework on top of JADE.

ParADE – Characteristics

- ParADE agents:
  - Integrate reactive and goal-directed behaviours to balance autonomy and efficiency;
  - Exploit the FIPA ACL with a minimalist semantics.
- ParADE generates Java code from:
  - Ontology diagram, models the part of the ontology that support the communication;
  - Architecture diagram, defines the architecture and the interaction protocols.
Example – Song Seller

```java
public class Shop extends ShopAgent {
    protected void init() {
        // Set the agent model
        Agent anyAgent = new AgentVariable("y");
        Song anySong = new SongVariable("w");

        // Plans to achieve intentions
        // If 'anyAgent' requests for a song and the song is available, then execute 'ActionBody'
        plan(available(me, anySong),
             done(sell(anyAgent, anySong)),
             new ActionBody() {
                public void body(Goal g) {
                    Done done = (Done)g;
                    Sell sell = (Sell)done.getAction();
                    sell.perform();
                    forget(intend(sell.getAgent(), done));
                    achieved(done);
                }
            });

        Song OneHeadlight = new ConcreteSong("One Headlight", 1000);
        believe(available(me, OneHeadlight));
        ConcreteAgent receiver = new ConcreteAgent(receiver);
        schedule(inform(receiver, available(me, OneHeadlight)));
    }
}
```

Implementing Multiagent Systems

- Inter-agent implementation aspects are orthogonal to intra-agent ones
  - Given a set of agents
    - With internal architecture
    - With specified interaction patterns
  - How can we glue them together?
    - Letting agents know each other
    - How to enable interactions?
    - Promoting spontaneous interoperability
    - How to rule interactions?
      - Preventing malicious or self-interested behaviours?

Multiagent Infrastructures

- Enabling and ruling interactions is mostly a matter of the infrastructure
- The "middleware" layer supporting communication and coordination activities
  - Not simply a passive layer
  - But a layer of communication and coordination "services"
    - Actively supporting the execution of interaction protocols
    - Providing for helping agents move in unknown worlds
    - Providing for proactively controlling, and possibly influencing interactions

Influence of the Infrastructure on MAS Analysis and Design

- Given that the infrastructure is somehow "active", what type of intelligence can it host?
  - Are there computational activities that can be delegated to the infrastructure?
    - E.g., finding other agents, re-shaping topology of interaction patterns, balancing load...
    - Or must these activities be in agents themselves?
  - How does this influence the design of a MAS?
    - When designing agents, should we know what the infrastructure can do?

Communication vs. Coordination Infrastructures

- Communication Infrastructures
  - Middleware layer mainly devoted to provide communication facilities
    - Routing messages, facilitators, etc.
    - FIPA defines a communication infrastructure
    - Communication enabling
- Coordination Infrastructure
  - Middleware layer mainly devoted to orchestrate interactions
    - Synchronization, and constraints on interactions
    - MARS and Tucson are coordination infrastructures
  - Activities ruling
Communication Infrastructure

- Agent in a MAS have to interact with each other, requiring
  - Finding other agents
    - Directory services in the infrastructure keep track of which agents are around, and what are their characteristics (e.g., services provided)
  - Re-routing message
    - Facilitator agents (parts of the infrastructure) can receive messages to be delivered to agents with specific characteristics, and re-route them
  - Control on ACL protocols
    - The execution of a single protocol can be controlled in terms of a finite state machine

Example of Communication Infrastructures: JADE (1)

- Implements a FIPA platform with all necessary services, e.g., DF.
- JADE
  - Is distributed across the network in terms of containers;
  - Provides management facilities, e.g., RMA;
  - Provides advanced development facilities, e.g., Sniffer.

Example of Communication Infrastructures: JADE (2)

- Interaction protocols are the FIPA way to manage interactions.
- JADE provides support for FIPA generic interaction protocols, e.g.:
  - FIPA Contract net;
  - FIPA English and Dutch auctions.
- JADE implements interaction protocols as FSM behaviours.

Features and Limitations of Communication Infrastructures

- There is not “application intelligence” in the infrastructure
  - The service provided are
    - Of a very general-purpose nature
    - Not re-configurable to meet the need of specific applications
- There is no global MAS orchestration
  - The only proactive control is on individual protocols
    - There is no way of controlling and influencing the global behaviour of a MAS
    - How to control self-interested behaviour, unpredictable dynamics, programming errors??
- This reflects in both advantages and drawbacks in multiagent systems engineering

Software Engineering with Communication Infrastructures

- All application problems are to be identified and designed in terms of
  - Internal agent behaviours and inter-agent interaction protocols
    - These include, from the intra-agent engineering viewpoint:
      - Controlling the global interactions
      - Controlling self-interested behaviours
- Advantages:
  - All in the system is an agent
  - The engineering of the system does not imply the engineering of the infrastructure
  - A standard has already emerged (FIPA)
- Drawbacks:
  - The design is hardly re-tunable
  - Global problems spread into internal agents’ code

Coordination Infrastructures

- The infrastructure is more than a support to communication
  - It can embed the “laws” to which interaction must obey
    - E.g., to specify which agents can execute which protocols and when
  - It can control the adherence of the MAS behaviour to the laws
    - E.g., to prevent malicious behaviours
  - Such laws can be re-configured depending on the application problem
    - E.g., English vs. Vickery auctions have different rules
Example of Coordination Infrastructures: MARS (1)

- Agents interact via a set of localized shared data space
  - One data space for each MAS
  - Or one data space for each Internet node
  - Data spaces mediate all interactions
- Such interaction can be
  - Stateless data exchanges
  - Stateful execution protocols
  - The data space acts as stateful repository of interaction messages
- The data space is active and programmable
  - It can proactively control and influence the interactions
  - On the basis of application-specific laws that can be re-configured at run-time

Example of Coordination Infrastructures: MARS (2)

- The data space can embed the coordination laws
  - Ruling, other than enabling, interactions
- Global control on the behavior of the MAS can be enacted
  - Interaction actions can be influenced and constrained
  - Control of self-interested behavior and errors
- Ease of maintenance
  - To change the behavior of the MAS, no need of changing agents, only coordination laws
  - e.g., from English to Vickery auction

Example of Coordination Infrastructures: Fishmarket

- Each agent in a MAS
  - Is dynamically attached a controller module
  - In charge of controlling its external actions (i.e., protocol execution)
- Inspired by real-world fish market auctions
  - Fishers participate in auctions by implicitly respecting local rules
- There is an implicit (institutional) control

Features and Limitations of Coordination Infrastructure

- The infrastructure
  - Provides for controlling the global behavior of the system
  - Can be re-configured to specific application needs
- This introduces problems of
  - In the case of open systems, is it correct to limit agents’ autonomy by constraining their behavior?
    - Who controls who?
  - Agents are no longer the only repository of “intelligence”
    - The infrastructure is intelligent, or at least active too
  - Increase of complexity?
  - Should we consider the infrastructure as an additional agent?

Software Engineering with Coordination Infrastructure (1)

- Clear separation of concerns
  - Intra-agent goals
  - Global MAS goals and global rules of the organizations
  - Such separation of concerns has to reflect in analysis and design
- Example: the Gaia methodology version 2
  - Explicitly tuned to open MAS
  - Implicitly assuming the presence of a coordination infrastructure
    - Identification of global organizational rules as a primary abstraction in the software process

Software Engineering with Coordination Infrastructure (2)

- Advantages
  - Separation of concerns reduces complexity in analysis and design
  - Inter-agent issues separated from intra-agent ones
  - Design for adaptivity perspective
    - Agents and rules can change independently
  - Intelligence in the infrastructure
    - A trend in the scenario of distributed computing
- Drawbacks
  - Implement both agents and infrastructural programs
  - Agents are no longer the only active components of the systems
    - No longer homogeneous
  - Lack of standardisation
Institutions

May basic researches in the area of MAS recognize that:

- Agents do not live and interact in a virgin world
  - Agents live in a society, and as that they have to respect the rules of a society
  - Agents live in an organization which can effectively executed only in respect of organizational patterns of interactions

In general: Multiagent systems represent institutions

- Where agents must conform to a set of expected behaviour in their interactions
- Such an approach requires the introduction of a conceptual coordination infrastructure during analysis and design (as in Gaia v. 2)

Summarizing Implementation Issues

(intra) Agent Implementation

- Different architectures available (OO vs. BDI)
- The choice may depend on
  - Ease of acceptance (OO easier to be accepted)
  - Application (for intelligent agents, BDI eases implementation)
- The methodology should abstract from implementation
  - Unfortunately, the design has to assume something about the internals of agent, reflecting in implementation mismatches

(inter) Multiagent Systems

- Communication vs. coordination architectures
- Intelligence reside only in agents or also in infrastructures?
- The choice may depend on
  - Ease of design and maintenance vs. ease of implementation
  - Designers attitudes
- However, institutions are a reality
  - Currently require a coordination infrastructure

Part 4

Research directions and visions

Mobility & Ubiquity

- The “Pervasive Computing” Umbrella
- Computing everywhere
  - Enabled by small portable devices
  - And low-costs wireless communications
- More than PDAs:
  - Embedded computing-based sensors
  - In cities, homes, cars, furnitures, clothes, bodies...
  - Locally connected in a global network
- Huge amounts
  - Dozens per persons, hundreds of billions in the world!
  - May be in need of coordinating with each other!

Mobility & Ubiquity: Engineering What?

- When billions of components may be potentially involved:
  - What does it mean to engineer a system?
    - What is the system (intrinsic openness)?
    - How can we control it (who controls what)?
    - How can we run it (the system is already and always running)?
- Top-down approach impossible:
  - Impossible to control the design of each component
  - Impossible to design and control the overall behavior of the system by controlling the global outcomes deriving from the interactions among its components
  - Impossible to “halt” the systems, modify it, and run it again accordingly to the new specifications

Part 4: Outline

- Open and promising research directions
  - Mobility & Ubiquity
  - Emergent Behaviour: Dynamic systems & Complexity
  - Self-organisation
  - Performance models
  (note: no solution sketched, mostly problems ;-)
**Bottom-up Engineering**

Starting from:
- Already available autonomous components (agents)
- Interacting with each other
- In a dynamic environment

How can we:
- Say what the global behavior will be?
- Influence "by design" such global behavior so as to guarantee that it will be as desired?
- Guarantee a specific behavior in spite of environmental dynamics?

Bottom up approaches:
- The study of "emergent behaviors"
- "Indirect control" of emergent behaviors

---

**Example of Emergent Behaviors: Cellular Automata**

- Shows the influence of environmental dynamics with a very simple model
- "Normal" cellular automata
  - State of simple cells determined by neighbor cells' states
  - No global patterns emerge

---

**Example of Emergent Behaviors: Dissipative Cellular Automata**

When the environment influences the state of cells:
- Global structures emerge
- Depending on the dynamics of the environment
  - The amount of "energy" flowing from the environment
  - Phenomenon similar to Prigogine's "dissipative structures"

---

**Controlling Emergent Behaviors in Dissipative Cellular Automata**

- By controlling a low percentage of cells
  - Either by changing the internal behaviors of a limited set of cells
  - Or by imposing a pre-determined state in a localized portion of cells

Do similar results apply to multiagent systems?
How to organize them into a methodology?

---

**Example: Indirect Engineering of Mobility**

- A multitude of "mobile agents" in an environment
  - Users, software modules, robots, etc...
- In need of orchestrating their movements
  - Avoid traffic jams, load balancing, meeting, etc.
- How can we globally coordinate such movements?
  - No control over each agent
  - Almost impossible to have each agent know all the other agents and decide where to go consequently:
    - Computational costs unbearable: global optimization algorithms too complex to be executed by each agent
    - Communication costs excessive: each agent should negotiate with the other agent its next movement
  - No Control: we cannot impose the execution of specific algorithms or of specific communication protocols to agents

---

**Engineering Mobility with Co-Fields**

- Global indirect control achieved via the environment (i.e., the infrastructure):
  - The infrastructure should be able to store and propagate simple (yet effective) contextual information
    - Determined by the position of the, as if it were a "gravitational field"
    - Suggesting agents on how to move
  - Globally coordinated behaviors achieved in a cheap way!
- Can similar approaches apply to areas different from mobility?
- How to approach that in a methodology?
Towards Complex Systems Engineering

- Traditional software systems:
  - Assumes control over components
  - Model software via formal systems

- Complex software systems:
  - No control over components
  - Requires different modeling approaches

- Novel modeling approaches:
  - Dynamical systems
  - Complex systems (self-organized criticality)

Dynamic Systems & Complexity

- Modeling software systems as made up of:
  - "mechanical" components
  - driven by environmental forces on the basis of observable properties (e.g., masses or electrical charge)
  - Moving in an abstract "phase space"

- Example: Co-Fields
  - The global movements can be determined by simple dynamical systems modeling

- General way of modeling still missing!

Self-Organized Systems

- Large systems of autonomous components organize in specific topological structures
  - Small worlds
  - Power-law networks

- These systems are in a state of so called "self-organized criticality"
  - They live at the edge of chaos
  - Exhibit peculiar dynamic properties

- Such general property applies to a large class of computational (non-engineered) systems:
  - Internet, Web, Gnutella

- Can a similar property impact on "engineered systems" (as multiagent systems are)?
  - Worth to be investigated!

Performance Models for Complex MAS

- What is "performance" in a complex MAS?
  - Traditional perspective on performance:
    - speed of execution, scalability, fault-tolerance
  - Does not necessarily apply in MAS
    - If execution is never ending, speed of execution becomes irrelevant, or should be re-formulated
    - Systems may be global by definition, so scalability may be a pre-condition
    - Autonomy of components and environmental dynamics make execution unpredictable and inaccurate, so that the concept of fault-tolerance should be re-formulated...
  - Novel performance models may be required
    - Delegation, trust, self-organization...

Adapting Performance Models for MAS

- Speed of execution
  - When MAS are forever running, speed of execution must become
    - Speed of reaction to an event in the environment, to a changed condition, to a request, to an
    - How can such local properties can translate in global properties of a MAS?

- Scalability
  - When systems are immersed in a global world, scalability must measure
    - The capability of the system to grow w.r.t to what?

- Fault-tolerance
  - A system that is imperfect by definition cannot break in an on-off way
  - We must measure the degradation of a system
  - Conversely, if the system can learn...

Need of Novel Performance Models for MAS

- Autonomy & Environmental dynamics
  - Performance of self-organizations
  - How long does it take for a system to re-organize
  - Does the re-organized system preserve the properties of the original one? How close?

- Trust
  - We delegate agents and MAS to do something
  - How much can we trust the behaviour of a MAS?
  - We must measure reliability in delegation to sell quality systems!

- Other performance issues we have missed?
  - Very likely to exists…..open issue!
CONCLUSIONS!

- In our humble opinion, agents will become the dominant paradigm in software engineering
  - AOSE abstractions and methodologies apply to a wide range of scenarios
- Several assessed research works already exist
  - Modeling work
  - Methodologies
  - Implementation Tools
- Still, there are a number of fascinating and largely unexplored open research directions…
  - Ubiquity, self-organization, performance…

Thank You!

Hope you enjoyed it!

Questions?

Contact Info

- Federico Bergenti
  bergenti@ce.unipr.it
  http://www.ce.unipr.it/people/bergenti
- Onn Shehory
  onn@il.ibm.com
  http://www.il.ibm.com
- Franco Zambonelli
  franco.zambonelli@unimo.it
  http://www.dsi.unimo.it/Zambonelli

Addendum

Selected References

- Introductory to Agents and Multiagent Systems

- Agent Abstractions are Everywhere!

- Selected References