Multiagent coordination:
A Computer Science perspective

Paolo Ciancarini

mailto://ciancarini@cs.unibo.it
http://www.unibo.it/~cianca

Dipartimento di Scienze dell'Informazione
University of Bologna - Italy

MAAMAW, Annecy, May 2001

Outline

Designing WWW-oriented multiagent applications

Active documents are agents

Towards agent-oriented software engineering

The role of coordination

The Internet as a programmable platform

“The computer is the network”: the convergence of Information and Communication Technologies produces new opportunities for industry, research, and teaching.

Example 1 The mp3 format and the Napster service are challenging the music industry: the old ways of distributing, selling, storing and playing music are obsolete

Example 2 Several Universities are now offering courses for designers, managers, and even art directors (eg. “Web DJ”) of WWW-based enterprises: we started forming people for the “contents industry” over the Internet

The convergence, or “networking”, of ICT industries is pushing the development of novel appliances, applications, services, and even organizational theories where the Internet plays a major role

Computer scientists and engineers are challenged to adapt themselves to the new platform, inventing new methods to expoit the new computing models enabled by the Internet

Remark:
we should probably start speaking of “Internet science” rather than “Computer science”

The Internet as a platform for groupware

An important application of the Internet is in the field of *groupware*, that is a domain offering interesting and important design problems

Groupware: document-centric applications which organize communities of users

- *mobility* of people, hosts, and documents
- *communication*: synchronous (“same time”) or asynchronous (“any time”)
- multi-user *interaction*: “same place” or “different places”
- *composition*: groupware is usually the result of the combination of several software technologies
- *agenthood*: in groupware several activies can be performed by autonomous programs (that are possibly mobile)
- *document-centric*: documents are complex data structures with user-specific contents, structure, and behaviours

WWW-based, agent oriented groupware

The software industry is redefining itself into a sw-intensive service industry: eg. Microsoft says that its main competitor is no more the software producer Oracle, but the Internet Service Provider America On Line.

Currently most sw-intensive services are actually Web applications supporting some form of groupware

Example: Microsoft Hailstorm

WWW-based, agent-oriented groupware is quite challenging:

⇒ software technologies related to the WWW are fastly evolving, usually as a result of some standardization process (by organizations like W3C or OMG)
⇒ classic software engineering techniques are unsuitable for Internet applications: in fact, we need novel network-aware agent-oriented ontologies and tools
⇒ these applications usually offer services to different organizations of the real world, which usually use different ontologies for documents, services, and agents

The role of network layers

Software technologies for Internet applications need novel, “network-aware” specification, design, and programming languages and tools

A key issue when an application includes “network-aware” (e.g. autonomous or mobile) components is how to design its *architecture*, which can be decomposed in at least three different layers:

- The *physical network* layer, made mostly of *immobile hosts* and reliable and fast connections, where a mobile entity consists of a piece of hardware using a connection usually unstable (e.g. wireless) and with low bandwidth

- The *middleware network* layer, made of *abstract machines* (e.g. a JavaVM, an XWindow server, etc.), where a mobile entity consists of a whole process or a service able to migrate from a host to another host

- The *logical network* layer, made of application code scattered over the middleware network, where a mobile entity consists of an *agent* able to move from an abstract machine to another one

Remark:
network layers are usually related to different organizational layers inside an organization, thus designing the logical layer of groupware actually means to design (the behavior of) a social organization

Mobile entities in the WWW

The original WWW was based on two separate concepts:

⇒HTTP servers distribute documents on demand to client browsers: this is *mobility of data* (HTML and XML are not Turing equivalent: they are just SGML dialects to specify data structures like paragraphs or tables)

⇒A browser can "navigate" through the hypertext links, requesting HTML pages to a server, and sometimes "jumping" from a server to another server. Although URLs denote static, "physical" resources, they can be passed around: this is *mobility of references*

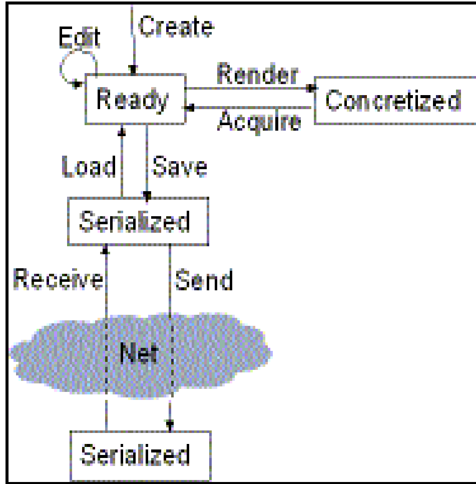
Remark: Mobility of reference means both that a channel name can be passed around, and that a process can detach a channel and connect to another channel

A third concept of mobile entity:

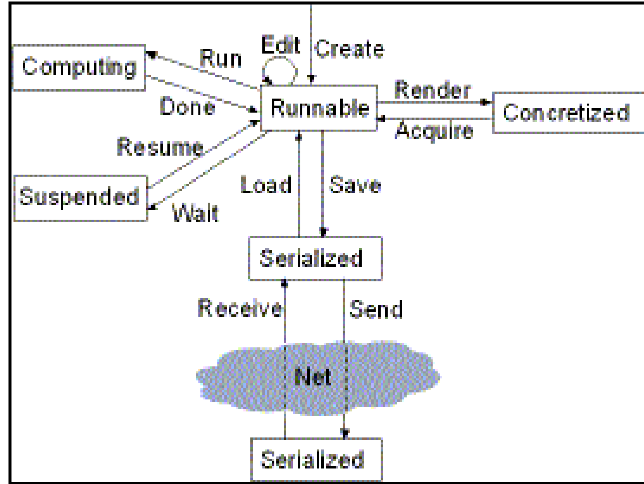
⇒An **active document** (= contents+structure+behavior) moves across the net, from servers to browsers and back

Technologies for active documents

Document lifecycles:



Passive document



Active document

An "active" document provides support for *interactions*; that is, it can include animations, perform computations, provide support for searching, etc.

Active-X objects, Java applets, JavaScript scripts or even complex PostScript or TeX programs can be used to build documents offering more than their content

The idea of "active document" consists of putting the behaviour together with the contents: using **generic markup** behaviors and actions are applied just like formatting

Remark: an active document is an (autonomous) entity including code and data, and moving around: it is an agent!

Representing documents

We use the term “document” with a broad scope, meaning any kind of data structure which network-aware applications can exchange

(passive) document: contents + (structured) representation

Example: RTF is a language of commands that a word processor has to apply to a document to render its contents, in terms of fonts, justification, margins, etc.

HTML has been invented to write passive documents to be displayed inside WWW browsers

(HTML) document: contents + procedural markup

Document processing applications use two different approaches to represent a document

a) a proprietary, binary, *machine-readable*, code:
   Examples: MS Word, Adobe PDF, SUN Java bytecode
b) an open (standard), ASCII-based, *human-readable* code:
   Examples: RTF, HTML, PostScript, TeX

Remark: ASCII-based documents are “flat”: their structure either is “wired” inside the applications which first parse then can manage them, or some further code (markup) is needed to give structure to an ASCII file

Procedural vs declarative markup

Formatters (eg. TeX or PostScript) assign a rendering behaviour to documents represented as files mixing formatting commands (*mark-ups*) and text

In order to build a “page”, formatters are driven by markup commands interspersed in the document text: formatters are in fact compilers

A system such as TeX produces high quality results because layout algorithms are able to approximate the behaviour of experienced professionals

However, TeX (and HTML) markups are very procedural, and this is bad for two reasons:

- the logical structure of a document is not expressed in the markup, thus searching document abstractions (eg. all titles, represented by indented bold lines) is difficult
- the concept of style is non existent, thus changes in style require revising all markup commands

A solution was the introduction of declarative markup languages (like LaTeX), useful to declare both the logical structure of a document and the styles to be used

Example

⇒`\documentstyle[twocolumn]{article}`

Declarative markup: SGML

As an a effort to develop a standard declarative markup language, in 1980 IBM proposed the Generalized Markup Language, which when adopted in 1986 by ISO became SGML (ISO/DIS 8879)

A SGML document is composed of three parts: the SGML declaration, the Document Type Declaration, and the document instance.

⇒SGML only specifies the structural elements composing a document
⇒SGML does not specify rendering semantics of its documents (this is a task for *stylesheets*)
⇒SGML does not provide support for hypertext links (meant for non-interactive uses)

SGML includes a meta-language to declare new tag types, to form a Document Type Declaration (DTD)

A SGML document contains elements, entities, comments and processing instructions (eg. the LINK marker is used to give behavioural semantics to declarative tags)

Towards active documents: XML

XML is a standard markup language (defined by W3C, and derived from SGML) to describe the *structure* of documents; instead, documents *behaviours* are specified using either stylesheet languages (e.g. XSL) or even programming languages (e.g. Java)

document: contents + structure + behaviours

Remark: declarative markup is either structural or semantic

Structure: A book is composed of chapters, sections, titles, notes, etc. A letter is composed of sender, addressee, salutations, body, signature, attachments, etc.

Semantics: A news item about a criminal act may specify the source of the news item itself, the description of a sequence of acts, the name of the place where the acts took place, the name and rank of the involved police officers, the stolen amount, etc.

What is a hypertext document

Definition: a *hypertext document* is defined by

- its contents,
- its structure,
- one or more "behaviours", and
- its relationships with other documents

Intuitively, a document carries some *information* and has some *structure*: a book, a report, a letter, a program are examples of documents of different forms

When documents live inside a computer they have a *physical representation* based on some data structure

When we consider a document in abstract, it is fully defined by its contents and logical structure: the structure of a document is an instance of a *document model*, that is an ontology of abstract entities suitable to describe the document's elements (eg. chapters, sections, paragraphs, pages, etc.)

Any document can display several *behaviours*:

⇒ a rendering, or presentation behaviour, defines how a document is displayed on an output device, like a screen or a printer (e.g. pretty printing is a rendering behaviour)

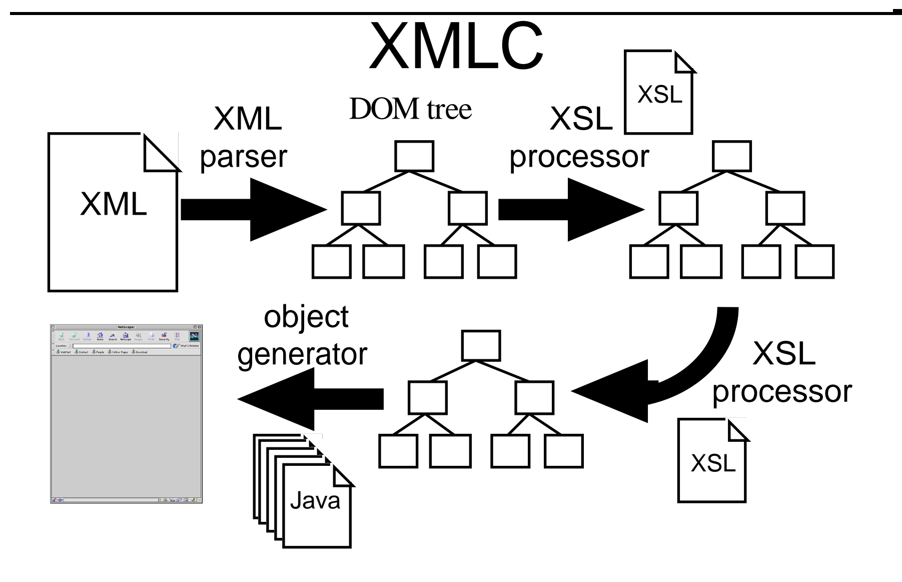
⇒ a view, or control behaviour, defines how a user can interact with a document (e.g. using hypertext)

⇒ a static semantics defines how a document can be analysed with respect to some verification rules

Active document are agents

XSL is not up to XML in terms of generality: specialized notations are not supported. At UniBologna, we have created an open set of formatting objects that are loaded when needed, depending on the required behavior of a document.

Each document has a stylesheet associated that maps its elements to the available formatting objects; each formatting object is then associated to a sw module (a JavaBean), that we call *displet*, displaying the information (we exploit the dynamic linking capabilities of Java)



Some applications that we have explored:

- Special typographical elements; layout management
- Notations for software engineering documents
- Management system for hypertext UML documents
- Porting of ToolBooks inside standard browsers
- DSS for financial applications
- WorkSpaces (a workflow management system)

Declaratively active documents

We have two reference architectures for displets, that we call "server-side" and "client-side", respectively

The *server-side* architecture is more efficient, but less flexible (a behaviour is pre-processed and "wired-in" a document, that then is sent to a browser and displayed)

The *client-side* architecture is a multiagent system that can be used to have documents performing activities, rather than just be displayed

Since we associate displets which are Java Beans to XML documents, we can ask a Bean to paint itself, or to perform any other method of its classes

Depending on the stylesheet and the Java classes, then the same document can behave in any of different ways

The code performing the activities is not part of the document (as in Active X or similar systems) but declaratively associated to the document

Example: Music scoresheets
We have defined a DTD for music scoresheets, thus we can represent textually any music score. Then we have defined stylesheets to display, animate, and play a score. Then we have enriched the displets able to display scores with editing capabilities, so that the original document can be modified

The impact of agent-hood

The WWW made clear that documents should be considered as portable, application-independent components requiring specific models and languages

However, the design of WWW-based groupware needs more than just a technology for sw components: documents are not only active; they are *interactive* agents (their users play some role) and moreover they activities need to be coordinated

Databases were the focus of application design in the '80
Components were the focus of application design in the '90
Agents are being the focus of application design in the '00

Component-based applications are usually built on top of a distributed middleware platform

⇒ CORBA (Common Object Request Broker Architecture)
⇒ Lotus Notes
⇒ World Wide Web
⇒ Sun's Jini
⇒ Microsoft .NET initiative

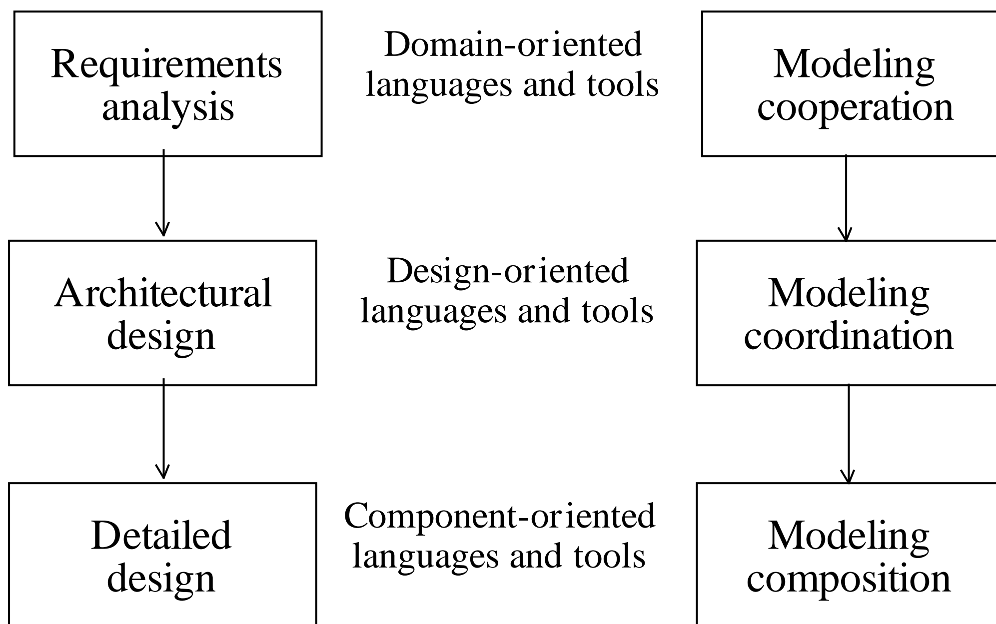
All these architectures offer some concept of *agent-hood*

Agent oriented sw engineering

Sw engineering deals with development processes and tools

"Classic" sw engineering is not adequate for the network, because it lacks of conceptual tools to deal with *interaction*

Classic vs. agent-oriented sw engineering process:



⇒ *Organizational model*: the abstract description and analysis of all roles involved in a system (cf. Use cases in UML)

⇒ *Coordination model*: the description of a sw architecture in terms of agents and their activities (eg. UML is weak in dealing with agent-oriented architectures)

⇒ *Composition model*: the set of mechanisms used to implement, reuse, or activate components and assign resources (eg. "JavaBeans components representing XML documents and roaming a web of HTTP servers")

A case study in Internet groupware

International scientific conferences are organised involving several people distributed all over the world
- authors of papers
- reviewers of papers
- Program Committee members
- Program Chairs (co-ordinators)
- conference organisers

The following activities have to be performed:
0. The "social laws" of conference management are stated
1. PC-members state their competence fields
2. Authors submit their papers by a given deadline
3. PC-chairs distribute papers according to the competence
4. Reviewers return they evaluations by a given deadline
5. PC-members decide according to the social laws
6. Authors of accepted papers prepare a final version
7. PC-chairs edit the proceedings
8. Organisers handle the list of talking/attending people

⇒Most people involved are mobile: scientists travel a lot
⇒Activities involve people cooperating on structured docs
⇒Most activities are routinary and can be managed by automatic scripts exploiting e-mail/ftp/WWW services

How should we design applications like a conference management system in terms of cooperation laws, coordination mechanisms, and component architectures?

Conference management over the Internet

Basic idea: submitted papers are agents (active documents)

Organizational model: *which social laws for roles and activity workflows?* we have to decide which conference organization we prefer:

single/multiple tracks and conference workflow;
tyrannical, oligarchic, democratic cooperation management;
authors of papers anonymous/known to reviewers; etc.

Coordination model: *how do we organise agents?*
we have to decide which sw architecture we offer to the agents, eg. a database-like or a peer-to-peer or a MUD-like one, and how agents interact with their environment

Composition model: *how do we reuse and compose components into architectures?*
we have to detail which components we allow and how do they "connect", eg. we could ask that all papers are based on PostScript (so that some browser could manage them) or on XML/XLink-Xpointer (so that they could form an hypertext and be managed by a search engine) or on .NET components

The impact of software architectures

The importance of studying software architectures and their related co-ordination models cannot be overestimated:

Napster is a service based on a peer-to-peer software architecture: its success is so big that it is redefining the music industry, and inspiring novel services and projects (eg. Sun's Peer-to Peer networking initiative)

In order to build a new generation of Internet-aware programming languages, we need to study and understand how co-ordination can be modelled and embodied in a software architecture

Agents and organisations

Internet-aware applications are designed by integrating several technologies; from a sw engineering perspective, we need methods and tools to master the complexity of such an integration

In my opinion the most interesting questions to be studied are in the field of specifying and analysing the organisations of agents, or organizational models. For instance, we need notations to describe and study e-commerce service chains, or logistic support systems

The increasing success of the concept of ERP shows that the introduction, or the adaptation, of Internet-aware services to a company reshapes its organisational forms and behaviours

We lack of notational and reasoning tools able to support the analysis of societies of agents, and the related organisation of services

Research on agent-based software is currently spread over several computer science sub-disciplines, like for instance Artificial Intelligence, Distributed System Programming, Network-aware Programming Languages, Information Retrieval Systems (a very partial list!)

A task for Agent-Oriented Software Engineering researchers: to develop a uniform framework including specific development process models, meta-models for co-operation, co-ordination, and composition, and tools and environments to support all the related design activities

Coordination

Coordination is a key concept for studying the activities of complex dynamic systems

⇒*Coordination is managing dependencies between activities* [Malone&Crowston 94]. All instances of coordination include *agents* performing *activities* that are *interdependent*

⇒*Coordination is the process of building programs by gluing together active pieces; a coordination model is the glue that binds separate activities into an ensemble*" [Carriero&Gelernter 92]

A coordination model is the formal basis (semantics) for a coordination language; usually a coordination language has to be combined with a conventional programming language to obtain a fully-fledged programming language

A number of co-ordination languages have been defined and studied in the last ten years; however the field is far from being exhausted, especially because the concept of "co-ordinable entity", or agent, has still to be fully understood

Coordination models

A *coordination model* is an formal framework useful to study and understand problems in designing programming languages and software architectures including several agents

In other words, a coordination model defines *how agents interact and how their interactions can be controlled*

This includes dynamic creation and destruction of agents, control of communication flows among agents, control of spatial distribution and mobility of agents, as well as synchronisation and distribution of actions over time

Coordination models differ mostly in the way they control interaction: for instance, different models could offer different kinds of **mobility**:

- *planned*: an agent's itinerary across some locations is statically predefined;
- *spontaneous:* an agent's itinerary is not statically predefined, but the next location is computed by the agent itself at run-time;
- *controllable:* a migration is forced by an authority in some location, using some I/O mechanisms to communicate with a remote agent. Interestingly, there are two types of controllable mobility: *sender-controlled* and *receiver-controlled.*

Different kinds of mobility require different coordination models

Communication, cooperation, coordination

	synchronous	**asynchronous**
communication	Messaging, chat	e-mail
cooperation	Napster	ftp, HTTP, WebDAV
coordination	distributed game playing (MUD), auction system	workflow (eg. conference management)

Communication mechanisms allow to exchange msgs and/or data streams; these are the basic services needed to build applications like IRC (Internet Relay Chat), e-mail, or teleconferencing systems

Cooperation mechanisms allow to share documents and resources; groupware applications like Napster and shared data repositories need specific cooperation protocols, respectively synchronous or asynchronous.

Coordination mechanisms allow the orchestration of multiple activities and services; sw platforms based on some coordination architecture are useful to build distributed game-playing environments or workflow support systems

Applications requiring coordination management: examples

Example 1:
Parallel simulation. Simulation models based on some notion of discrete, backtrackable time, require specific coordination techniques for a parallel implementation

Example 2:
Integration of whole applications, reconfiguring and coordinating the computations of several independent decision support systems (eg. inter-related spreadsheets under the control of different users)

Example 3:
Multiagent symbolic computing, as in an environment integrating theorem proving with model checking and other reasoning tools

Example 4:
Workflow systems for active documents. Modern document management systems are usually based on a notion of document-agent whose interactions with users, tools, and other documents-agents have to be explicitly managed

Coordinable software architectures

Examples of software architectures requiring coordination are
- the master-worker
- the client-server
- the peer-to-peer
- the software pipeline
- the blackboard
- the shared repository

We say that these architectures require coordination because their basci entities (the *coordinables*) are arranged in some special, well-defined, reusable structures including several components requiring some specific interaction protocols

Currently, software designers usually design a (coordinable) software architecture using low-level communication primitives and/or special module configuration languages

Instead, we suggest that a software designer should have clear a coordination model, embedded in a coordination language, to implement a specific software architecture

Coordination models

A *coordination model* offers mechanisms to control agent creation/connection/termination, and simple abstractions to define the semantics of connectors

Definition:
A *coordination model* is a triple (E,M,L), where:

- E are the *coordinable entities* (components):
  these are the agents which are coordinated. Ideally, these are the building blocks of a coordination architecture (eg. agents, processes, tuples, atoms, etc.)
- M are the *coordinating media (connectors)*:
  these are the coordinators of interagent entities. They also serve to aggregate a set of agents to form a *configuration*. (eg. channels, shared variables, tuple spaces, bags)
- L are the *coordination laws* ruling actions by coordinable entities wrt the coordination media. Usually the laws define the semantics of a number of coordination mechanisms that can be added to a host language

Some dimensions of coordination

The basic ideas in all coordination models and languages are "*minimalism*"(a small set of coordination primitives should suffice) and "*optimizability*" (it should be possible to reason on and compile coordination primitives)

Coordination models and related languages can be classified along a number of dimensions:

- location-less vs locality-based (named) coordination media
- transactional (multiset based) vs asynchronous (tuple based) coordination media
- procedural (imperative, functional, or logic) vs object-oriented (or agent-oriented) coordinables
- centralized vs decentralized coordination laws
- data-driven vs event-driven coordination primitives

Example: coordination and mobility

The issue of coordination mechanisms for mobile agents has been studied expecially in the context of environments for network-aware programming

Multiple Tuple Spaces are natural coordination media for both mobile code (eg. Java) and mobile agents (code+state)

The following proposals differ in the coordinable entities,
in the mechanisms used to access the tuple spaces, and
in the possibility of extending the coordination laws

System	*Coordinables*	*Media*	*Language*	*Laws*
JavaSpaces	JavaAgents	TS	Java+Linda	Fixed
Lime	MobileAgents	Local and global TS	Linda+asynch primitives	Fixed
MARS	Java Agents	Network aware TS	Java+Linda	Programmabl e (in Java)
PageSpace	Agents+ Services	Network unaware TS	Java+Linda	Fixed
TSpaces	Agents	Network unaware TS	Linda+ user-defined primitives	Programmabl e (Overriding)
TUCSON	Information agents	Network aware TS	Logic oriented Linda	Programmabl e (in Prolog)
WCL	Agents, applications	Automatic reallocation	Linda+asynch primitives	Fixed

See `ftp://ftp.cs.unibo.it/cianca/slides/rif.ps.gz`
for a reference list

Conclusions

Coordination models are a new exciting research field

Coordination models are an important tool for a new class of distributed applications, in which several agents (humans, tools, programs) have to be coordinated

There are several questions that are being addressed:

⇒ which coordination mechanisms are more expressive and useful?

⇒ which semantic models should be used to study such mechanisms?

⇒ which implementation techniques are best?

⇒ which software architectures match some specific coordination requirements?

⇒ which programming logics can be used to reason about coordination programs?

⇒ which new applications can be developed, which exploit the new technology?

A roadmap

Internet scientists and practitioners have to follow closely what organizations for "network governance" like W3C or OMG develop and define as software standards

In the next five years, we believe that:

⇒ the family of XML technologies will have a strong impact on sw technologies, programming languages and tools

⇒ the concepts of agent-oriented software engineering will gain momentum, possibly succeeding object-oriented software engineering as the dominant design method

⇒ "Active document"-centric software architectures will substitute client-server architectures

Conclusions

Network-aware, agent-based applications and services are designed by integrating several technologies; from a sw engineering perspective, we need methods and tools to master the complexity of such an integration and to study their impact on social organizations

Coordination and composition models are important concepts for agent-oriented software engineering; we nedd to improve our understanding of organizational models

However, we have a long way before understanding their inter-relationship

At present we are especially interested in:

⇒ developing and evaluating network-aware models and languages for agent-documents and related software architectures

⇒ developing formal specification languages useful to design software architectures including mobile components

⇒ building a Web-based software architecture able to support active documents (integrating XML and Java)