LECTURE 10: METHODOLOGIES

An Introduction to Multiagent Systems

http://www.csc.liv.ac.uk/˜mjw/pubs/imas/

1 Pitfalls of Agent Development

- *Lots* of (single and multi-) agent projects . . . but agent-oriented *development* recvd little attention.
- We now consider *pragmatics* of AO software projects.
- Identifies key *pitfalls*.
- Seven categories:
  - political;
  - management;
  - conceptual;
  - analysis and design;
  - micro (agent) level;
  - macro (society) level;
  - implementation.

1.1 You Oversell Agents

- Agents are *not* magic!
- If you can't do it with ordinary software, you probably can't do it with agents.
- No evidence that any system developed using agent technology could not have been built just as easily using non-agent techniques.
- Agents *may* make it easier to solve certain classes of problems . . . but they do not make the impossible possible.
- Agents are not AI by a back door.
- Don't equate agents and AI.

1.2 You Get Religious

- Agents have been used in a wide range of applications, but they are not a universal solution.
- For many applications, conventional software paradigms (e.g., OO) are more appropriate.
- Given a problem for which an agent and a non-agent approach appear equally good, prefer non-agent solution!
- In summary: danger of believing that agents are the right solution to *every* problem.
- Other form of dogma: believing in your agent definition.

1.3 Don't Know Why You Want Agents

- Agents = new technology = lots of hype!
  "Agents will generate US$2.6 billion in revenue by the year 2000"
- Managerial reaction:
  "we can get 10% of that".
- Managers often propose agent projects without having clear idea about what "having agents" will buy them.
- No *business plan* for the project:
  - pure research?
  - technology vendor?
  - solutions vendor?
  - ...

- Often, projects *appear* to be going well. ("We have agents!") But no vision about where to *go* with them.
- The lesson: understand your reasons for attempting an agent development project, and what you expect to gain from it.

1.4 Don't Know What Agents Are Good For

- Having developed some agent technology, you search for an application to use them.
- Putting the cart before the horse!
- Leads to mismatches/dissatisfaction
- The lesson: be sure you understand how and where your new technology may be most usefully applied.
  Do not attempt to apply it to arbitrary problems & resist temptation to apply it to every problem.

1.5 Generic Solutions to 1-Off Problems

- The "yet another agent testbed" syndrome.
- Devising an architecture or testbed that supposedly enables a range agent systems to be built, when you really need a one-off system.
- Re-use is difficult to attain unless development is undertaken for a close knit range of problems with similar characteristics.
- General solutions are more difficult and more costly to develop, often need tailoring to different applications.

1.6 Confuse Prototypes with Systems

- Prototypes are easy (particularly with nice GUI builders!)
- Field tested production systems are hard.
- Process of scaling up from single-machine multi-threaded Java app to multi-user system *much* harder than it appears.

1.7 Believe Agents = Silver Bullet

- Holy grail of software engineering is a "silver bullet": a order of magnitude improvement in software development.
- Technologies promoted as the silver bullet:
  - COBOL :-)
  - automatic programming;
  - expert systems;
  - graphical programming;
  - formal methods (!)
- Agent technology is *not* a silver bullet.
- Good reasons to believe that agents are useful way of tackling some problems.
- But these arguments largely untested in practice.

- Useful developments in software engineering: *abstractions*.

  Agents are another abstraction.

1.8 Confuse Buzzwords & Concepts

- The idea of an agent is extremely intuitive.
- Encourages developers to believe that they understand concepts when they do not.

  (The AI & party syndrome: everyone has an opinion. However uninformed.)
- Good example: the belief-desire-intention (BDI) model.
  - theory of human practical reasoning (Bratman et al);
  - agent architectures (PRS, dMARS, . . .);
  - serious applications (NASA, . . .);
  - logic of practical reasoning (Rao & Georgeff).
- Label "BDI" now been applied to WWW pages/perl scripts.

- “Our system is a BDI system” . . . implication that this is like being a computer with 64MB memory: a quantifiable property, with measurable associated benefits.

1.9 Forget it’s *Software*

- Developing *any* agent system is essentially experimentation. No tried and trusted techniques
- This encourages developers to forget they are developing *software*!
- Project plans focus on the agenty bits.
- Mundane software engineering (requirements analysis, specification, design, verification, testing) is forgotten.
- Result a foregone conclusion: project flounders, not because agent problems, but because basic software engineering ignored.
- Fequent justification: software engineering for agent systems is none-existent.

- But almost *any* principled software development technique is better than none.

Forget its *distributed*

- Distributed systems = one of the most complex classes of computer system to design and implement.
- Multi-agent systems tend to be distributed!
- Problems of distribution do not go away, just because a system is agent-based.
- Typical multi-agent system will be *more* complex than a typical distributed system.
- Recognise distributed systems problems.
- Make use of DS expertise.

1.10 Don't Exploit Related Technology

- In any agent system, percentage of the system that is agent-specific is comparatively small.
- The *raising bread model* of Winston.
- Therefore important that conventional technologies and techniques are exploited wherever possible.
- Don't reinvent the wheel. (Yet another communication framework.)
- Exploitation of related technology:
  - speeds up development;
  - avoids re-inventing wheel;
  - focusses effort on agent component.
- Example: CORBA.

1.11 Don't exploit concurrency

- Many ways of cutting up any problem.

  Examples: decompose along functional, organisational, physical, or resource related lines.
- One of the most obvious features of a poor multi-agent design is that the amount of concurrent problem solving is comparatively small or even in extreme cases non-existent.
- Serial processing in distributed system!
- Only ever a single thread of control: concurrency, one of the most important potential advantages of multi-agent solutions not exploited.
- If you don't exploit concurrency, why have an agent solution?

1.12 Want Your Own Architecture

- Agent architectures: designs for building agents.
- Many agent architectures have been proposed over the years.
- Great temptation to imagine you need your own.
- Driving forces behind this belief:
  - "not designed here" mindset;
  - intellectual property.
- Problems:
  - architecture development takes years;
  - no clear payback.
- Recommendation: buy one, take one off the shelf, or do without.

1.13 Think Your Architecture is Generic

- If you *do* develop an architecture, resist temptation to believe it is generic.
- Leads one to apply an architecture to problem for which it is patently unsuited.
- Different architectures good for different problems.
- Any architecture that is truly generic is by definition not an architecture . . .
- If you have developed an architecture that has successfully been applied to some particular problem, understand *why* it succeeded with that particular problem.
- Only apply the architecture to problems with similar characteristics.

1.14 Use Too Much AI

- Temptation to focus on the agent specific aspects of the application.
- Result: an agent framework too overburdened with experimental AI techniques to be usable.
- Fuelled by “feature envy”, where one reads about agents that have the ability to learn, plan, talk, sing, dance...
- Resist the temptation to believe such features are essential in your agent system.
- The lesson: build agents with a minimum of AI; as success is obtained with such systems, progressively evolve them into richer systems.
- What Etzioni calls “useful first” strategy.

1.15 Not Enough AI

- Don’t call your on-off switch an agent!
- Be realistic: it is becoming common to find everyday distributed systems referred to as multi-agent systems.
- Another common example: referring to WWW pages that have any behind the scenes processing as “agents”.
- Problems:
  - lead to the term “agent” losing *any* meaning;
  - raises expectations of software recipients
  - leads to cynicism on the part of software developers

1.16 See agents everywhere

- “Pure” A-O system = everything is an agent!
  Agents for addition, subtraction, ...
- Naively viewing *everything* as an agent is inappropiate.
- Choose the right *grain size*.
- More than 10 agents = big system.

1.17 Too Many Agents

- Agents don’t have to be complex to generate complex behaviour.
- Large number of agents:
  - *emergent functionality*;
  - *chaotic behaviour*.
- Lessons:
  - keep interactions to a minimum;
  - keep protocols simple;

1.18 Too few agents

- Some designers imagine a separate agent for every possible task.
- Others don't recognise value of a multi-agent approach at all.
- One "all powerful" agent.
- Result is like OO program with 1 class.
- Fails software engineering test of *coherence*.

1.19 Implementing infrastructure

- There are no widely-used software platforms for developing agent systems.
- Such platforms would provide all the basic infrastructure required to create a multi-agent system.
- The result: everyone builds there own.
- By the time this is developed, project resources gone!
- No effort devoted to agent-specifics.

1.20 System is anarchic

- Cannot simply bundle a group of agents together.
- Most agent systems require system-level engineering.
- For large systems, or for systems in which the society is supposed to act with some commonality of purpose, this is particularly true.
- Organisation structure (even in the form of formal communication channels) is essential.

1.21 Confuse simulated with real parallelism

- Every multi-agent system starts life on a single computer.

  Agents are often implemented as UNIX processes, lightweight processes in C, or JAVA threads.
- A tendency to assume that results obtained with simulated distribution will immediately scale up to *real* distribution.
- A dangerous fallacy: distributed systems are an *order of magnitude* more difficult to design, implement, test, debug, and manage.
- Many practical problems in building distributed systems, from mundane to research level.
- With simulated distribution, there is the possibility of centralised control; in truly distributed systems, such centralised control is not possible.

1.22 The *tabula rasa*

- When building systems using new technology, often an assumption that it is necessary to start from a "blank slate".
- Often, most important components of a software system will be *legacy*:

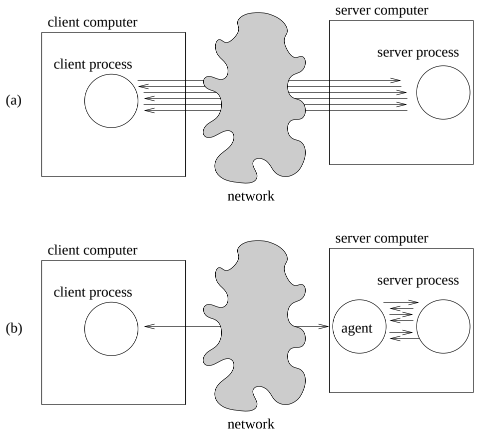
  functionally essential, but technologically obsolete software components, which cannot readily be rebuilt.
- Such systems often mission critical.
- When proposing a new software solution, essential to work *with* such components
- They can be incorporated into an agent system by *wrapping* them with an *agent layer*.

1.23 Ignore *de facto* standards

- There are no established agent standards.
- Developers often believe they have no choice but to design and build all agent-specific components from scratch.
- But here *are* some *de facto* standards.
- Examples:
  - CORBA;
  - HTML;
  - KQML;
  - FIPA.

2 Mobile Agents

- Remote procedure calls (a) versus mobile agents (b):



- Why mobile agents?
  - low-bandwidth networks (hand-held PDAs, such as NEWTON);
  - efficient use of network resources.
- There are *many* issues that need to be addressed when building software tools that can support mobile agents. . .
  - security for hosts and agents;
  - heterogeneity of hosts;
  - dynamic linking.

Security for Hosts

We do not want to execute foreign programs on our machine, as this would present enormous *security* risks:

- If the agent programming language supports pointers, then there is the danger of agents corrupting the address space of the host $\Rightarrow$ many agent languages don't have pointers!
- UNIX-like access rights on host;
- safe libraries for access to filestore, process space, etc;
- some actions (e.g., sending mail) are harmless in some circumstances, but dangerous in others — how to tell?

- some agent languages (e.g., TELESCRIPT) provide limits on the amount of e.g., memory & processor time that an agent can access;
- *secure co-processors* are a solution — have a physically separate processor on which the agent is run, such that the processor is in 'quarantine' ('padded cell').

Some agent languages allow security properties of an agent to be verified on receipt.
Hosts must handle crashed programs cleanly — what do you tell an owner when their agent crashes?
Trusted agents?

Security for Agents

- Agents have a right to privacy!
- We often do not want to send out our programs, as to do so: might enable the recipient to determine its purpose, and hence our intent.
- The agent might be modified (sabotaged!) in some way, without its owners knowledge or approval.
- An agent can be protected in transit by using conventional encryption techniques (e.g., PGP).
- In order to ensure that an agent is not tampered with, it is possible to use *digital watermarks* — rather like check digits.

Heterogeneity of Hosts

- Unless we are happy for our agents to be executed on just one type of machine (Mac, PC, SPARC, . . .), then we must provide facilities for executing the same agent on many different types of machine.
- This implies:
  - *interpreted language*:
    compiled languages imply reduction to machine code, which is clearly system dependent — reduced efficiency; (perhaps use virtual machine technology);
  - *dynamic linking*:
    libraries that access local resources must provide a common interface to different environments.

A Typology for Mobile Agents

- We can divide mobile agents into at least three types:
  - *autonomous*;
  - *on-demand*;
  - '*active mail*'-type

Autonomous Mobile Agents

- By *autonomous* mobile, we mean agents that are able to *decide for themselves* where to go, when, and what to do when they get there (subject to certain resource constraints, e.g., how much 'emoney' they can spend.
- Such agents are generally programmed in a special language that provides a `go` instruction... best known example is TELESCRIPT.

On-Demand Mobility

- The idea here is that a host is only required to execute an agent when it explicitly demands the agent.
- The best known example of such functionality is that provided by the JAVA language, as embedded within `html`.
- A user with a JAVA-compatible browser (e.g., NETSCAPE 2.0) can request `html` pages that contain *applets* – small programs implemented in the JAVA language.
- These applets are downloaded along with all other images, text, forms, etc., on the page, and, once downloaded, are executed on the user's machine.
- JAVA itself is a general purpose, C/C++ like programming language, (that does not have pointers!)

'Active-Mail' Agents

- The idea here is to 'piggy-back' agent programs onto mail.
- The best-known example of this work is the *mime* extension to email, allowing Safe-Tcl scripts to be sent.
- When email is received, the 'agent' is unpacked, and the script executed... hence the email is no longer passive, but *active*.

2.1 Telescript

- TELESCRIPT was a language-based environment for constructing mobile agent systems.
- TELESCRIPT technology is the name given by General Magic to a family of concepts and techniques they have developed to underpin their products.
- There are two key concepts in TELESCRIPT technology:
  - *places*; and
  - *agents*.
- Places are *virtual locations* occupied by agents. A place may correspond to a single machine, or a family of machines.

- Agents are the providers and consumers of goods in the *electronic marketplace* applications that TELESCRIPT was developed to support.
- Agents are interpreted programs, rather like TCL.
- Agents are *mobile* — they are able to move from one place to another, in which case their program and state are encoded and transmitted across a network to another place, where execution recommences.
- In order to travel across the network, an agent uses a *ticket*, which specifies the parameters of its journey:
  - destination;
  - completion time.

- Agents can communicate with one-another:
  - if they occupy different places, then they can connect across a network;
  - if they occupy the same location, then they can *meet* one another.

- TELESCRIPT agents have an associated *permit*, which specifies:
  - what the agent can do (e.g., limitations on travel);
  - what resources the agent can use.
- The most important resources are:
  - ‘money’, measured in ‘teleclicks’ (which correspond to real money);
  - lifetime (measured in seconds);
  - size (measured in bytes).
- Agents and places are executed by an *engine*.
- An engine is a kind of agent operating system — agents correspond to operating system processes.

- Just as operating systems can limit the access provided to a process (e.g., in UNIX, via access rights), so an engine limits the way an agent can access its environment.

- Engines continually monitor agent's resource consumption, and kill agents that exceed thei limit.
- Engines provide (C/C++) links to other applications via *application program interfaces* (APIs).
- Agents and places are programmed using the TELESCRIPT language:
  - pure object oriented language — everything is an object — apparently based on SMALLTALK;
  - interpreted;
  - two levels — high (the 'visible' language), and low (a semi-compiled language for efficient execution);
  - a 'process' class, of which 'agent' and 'place' are sub-classes;
  - persistent;

- General Magic claim that the sophisticated built in communications services make TELESCRIPT ideal for agent applications!

- Summary:
  - a rich set of primitives for building distributed applications, with a fairly powerful notion of agency;
  - agents are ultimately interpreted programs;
  - no notion of strong agency!
  - likely to have a significant impact (support from Apple, AT&T, Motorola, Philips, Sony).
  - not heard of anyone who has yet actually *used* it!

2.2 TCL/TK and Scripting Languages

- The (free) Tool Control Language (TCL — pronounced 'tickle') and its companion TK, are now often mentioned in connection with agent based systems.
- TCL was primarily intended as a standard *command language* — lots of applications provide such languages, (databases, spreadsheets, . . .), but every time a new application is developed, a new command language must be as well.

  TCL provides the facilities to easily implement your own command language.
- TK is an X window based widget toolkit — it provides facilities for making GUI features such as buttons, labels, text and graphic windows (much like other X widget sets).

  TK also provides powerful facilities for interprocess communication, via the exchange of TCL scripts.

- TCL/TK combined, make an attractive and simple to use GUI development tool; however, they have features that make them much more interesting:
  - TCL it is an *interpreted language*;
  - TCL is *extendable* — it provides a core set of primitives, implemented in C/C++, and allows the user to build on these as required;
  - TCL/TK can be *embedded* — the interpreter itself is available as C++ code, which can be embedded in an application, and can itself be extended.

- TCL programs are called *scripts*.
- TCL scripts have many of the properties that UNIX shell scripts have:
  - they are plain text programs, that contain control structures (iteration, sequence, selection) and data structures (e.g., variables, lists, and arrays) just like a normal programming language;
  - they can be executed by a shell program (`tclsh` or `wish`);
  - they can call up various other programs and obtain results from these programs (cf. procedure calls).
- As TCL programs are *interpreted*, they are very much easier to prototype and debug than compiled languages like C/C++ — they also provide more powerful control constructs. . .
  - . . . but this power comes at the expense of speed.

  - Also, the structuring constructs provided by TCL leave something to be desired.

- So where does the idea of an agent come in?

  It is easy to build applications where TCL scripts are exchanged across a network, and executed on remote machines.

  Thus TCL scripts become sort of agents.
- A key issue is *safety*. You don't want to provide someone elses script with the full access to your computer that an ordinary scripting language (e.g., `csh`) provides.
- This led to Safe TCL, which provides mechanisms for limiting the access provided to a script.

  Example: Safe TCL control the access that a script has to the UI, by placing limits on the number of times a window can be modified by a script.
- But the safety issue has not yet been fully resolved in TCL. This limits its attractiveness as an agent programming environment.

- Summary:
  - TCL/TK provide a rich environment for building language-based applications, particularly GUI-based ones.
  - But they are not/were not intended as agent programming environments.
  - The core primitives may be used for building agent programming environments — the source code is free, stable, well-designed, and easily modified.