Communicating Agents: An Emerging Approach for Distributed Heterogeneous Systems

Stephen Cranefield[1]
Martin Purvis
Department of Information Science
University of Otago

Paul Gorman
Department of Computer Science
University of Otago

July 1995

Abstract

The concept of an intelligent software agent has emerged from its origins in artificial intelligence laboratories to become an important basis for the development of distributed systems in the mainstream computer science community. This paper provides a review of some of the ideas behind the intelligent agent approach and addresses the question "what is an agent?" Some principal application areas for agent-based computing are outlined and related research programmes at the University of Otago are discussed.

[1] Address correspondence to: Dr S.J.S. Cranefield, Lecturer, Department of Information Science, University of Otago, P.O. Box 56, Dunedin, New Zealand. Fax: +64 3 479 8311 Email: scranefield@commerce.otago.ac.nz

1 Introduction

In recent years terms such as "intelligent agent", "software agent" and "agent-based computing" have been appearing with increasing regularity in conference papers, trade journals, software product announcements and even the popular press. With phrases such as "the next significant breakthrough in software development" (Sargent, 1992), "the new revolution in software" (Guilfoyle, 1994) and "the new software high ground" (Ousterhout, 1995), this technology is being hailed as having huge commercial potential both to the software industry and to the providers of information on the Internet.

In addition, researchers studying organisational structures and other complex systems have discovered that the notion of an "autonomous communicating agent" provides a natural metaphor for modelling system components and their interactions. There is currently much research under way into formalisms for representing and reasoning about agents' knowledge, languages for inter-agent communication and protocols for negotiation and cooperation between service-providing agents.

This paper looks at the reasons for the rise of the agent metaphor in software development and system modelling and discusses the question of what exactly constitutes an agent (*i.e.* what sort of system or software component could meaningfully be described by that term). Some principal application areas for agent-based computing are outlined and several research projects under way at the University of Otago are discussed.

2 The Rise of the Agent Metaphor

In the last ten years there has been a huge growth in the power and affordability of computers and the communications technologies to support them. As a result computers can now be found on almost every office worker's desk and in many homes. This has allowed organisations' problem-solving expertise (both human and computer-based) to become decentralised, with much decision making being performed through the communication and cooperation of physically separated individuals or teams of workers, each with their own specific area of expertise and store of domain-specific knowledge. In addition, each site involved in a collaborative project may be using different computing hardware, software and decision-making processes. This naturally leads to a view of remote sites as problem-solving or service providing "agents" whose exact internal structure is unknown.

For workers and home consumers alike, the exponential growth in hosts connected to the Internet[2] means there is a wealth of information and services available from the desktop, if only the relevant information can be located and filtered out from the vast mass of irrelevant material. The number of hosts on the Internet is currently growing at a rate of 15 to 20 percent a month (Malamud, 1995). The release of new operating systems with built-in Internet access software (such as Windows 95) will ensure this growth continues. Every day, more information is potentially available from an

[2] "It has been estimated that there are over 32 million people on the Internet and that this number has increased by 80% during the last year with one million new hosts added during the first six months of 1994" (Nejmeh, 1994).

Internet-connected worker's desktop. There is a desperate need for sophisticated software tools that navigate the Internet acting on behalf of the user, searching for the vital gems of information that match his or her needs and interests.

The consumer service sector is also responding to the growing number of people with Internet access[3] . For example, AT&T are planning an on-line service, PersonaLink, that will allow consumers to construct agents that can travel across networks to perform complex enquiries and transactions on service-providing hosts (Hill, 1994). PersonaLink, based on General Magic's Telescript language (White, 1994), will allow the creation of agents that lodge themselves on an information provider's machine (for a suitable fee) to keep watch and notify the user when a particular event is triggered (such as the value of the user's shares reaching a certain price on the stock market). Other agents could be designed to visit virtual shops and find the best price for a particular item, or to obtain the best available tickets for an opera without the user having to wait in a queue and explain their preferences at the booking office.

3 What is an Agent?

Due perhaps to its recent elevation to buzz-word status, many writers and developers of software utilities have been using the word "agent" to describe tools that perform relatively simple, albeit useful, tasks, while others feel that the term should be reserved for only the more "intelligent" applications. To clarify these matters and to avoid continuing to rely on some unreliable shared notion of what an "agent" is, some researchers are beginning to characterise the properties and tasks that are characteristic of an agent (Wooldridge and Jennings, 1995; Genesereth and Ketchpel, 1994; Foner, 1993). If we are to examine this problem, we might first consider why the notion of an agent is attractive. This in turn leads to two different views motivating the use of the term "agent".

1. Consider how the modelling of any complex dynamic system is initially performed in the everyday world. We observe that

   (a) it is natural to conceptualise the relevant features, *i.e.* the behavioural components to be modelled, in terms of simple or familiar elements; and

   (b) the overall model structure and the number of individual elements *must* be kept simple enough so that the entire model can be easily understood, manipulated, and modified, if necessary.

If the system to be modelled is significantly complex, restriction *b* means that the individual elements must, themselves, encompass a fair degree of complexity. In that case it is appropriate to express these complex elements in terms of those dynamic entities from the real world with which we are most familiar—human agents, and we intuitively construct mental models of this nature all the time. Software agents enable us to map these agent mental

[3] In 1993 this was by far the fastest growing commercial use of the Internet with a growth rate of 27.8% (Hill, 1994).

models directly into a computer representation, and consequently they facilitate the development of bigger and more complex systems. Systems developed by proponents of this approach are likely to have an agent-oriented architecture. We can refer to agents from the perspective of those who develop systems as *software engineering agents*.

2. End-users, too, (unconsciously) build mental models of the computer systems they use, and their satisfaction will be enhanced if their own mental model matches the system's behaviour. For relatively complex software systems, then, it can be advantageous to display interactive components that appear to the user as individual agents: these agents are likely to facilitate the rapid formation of the user's mental model of how the software works. From this perspective, software agents are appealing to *end-users* of systems. Systems developed by proponents of this second approach are likely to have an agent-oriented user interface. We can refer to agents from the perspective of the end-user as *user interface agents*.

The above two views of agents (which relate to how the agent idea is put to use) are not necessarily in conflict, but they can lead to some confusion about the nature of agents if the distinctions are not articulated. However, we can identify some essential properties of agents that are common to all approaches (Wooldridge and Jennings, 1995):

• *autonomy:* agents operate without direct (step-by-step) control of their actions from the outside;

• *goals or intentions:* agents are perceived to have goals that they attempt to achieve and may achieve these goals in various ways;

• *memory:* agents can remember past events (they retain state) and can possibly improve their behaviour as a result of this memory;

• *reactivity:* agents perceive their environment and react to changes in it or to direct actions upon them;

• *proactivity:* agents not only respond to changes in their environment, they are able to take the initiative and undertake actions in pursuit of their goals[4] .

For software engineering agents the above elements are explicitly represented in the make-up of the agents. For user interface agents, on the other hand, the above elements need only *appear* to the end-user to be present.

Many researchers attribute additional properties to agents and hold them to be essential. The AI research community, for example, is actively exploring elaborations to the basic notion of an agent outlined above. Apart from more extensive planning capabilities for agents, which has long been an interest in AI circles, researchers have

[4] For example, a user's personal mailbox agent when receiving an announcement of a new release of some public domain software might fetch the relevant files without further instruction.

been investigating additional mentalistic notions, such as belief and obligation (Shoham, 1993), and even emotional states (Bates, 1994). For the most part these elaborations appear to be oriented towards software engineering agents, but could be applied to user interface agents as well.

Researchers investigating user interface agents are concerned with the end-user's suspension of disbelief and add their own basic properties to agents. For example, Foner (1993) lists the following additional agent properties, which deal with agent-human interactions, as requisite:

- *personalizability:* agents should be able to develop their own individual ways of doing things;

- *discourse:* agents should be able to carry out a conversation with the end-user, from which can emerge something resembling a contract;

- *risk and trust:* we have to trust the agent with some task;

- *domain of discourse:* there must be an understood domain of discourse;

- *cooperation:* the agent must be perceived as something that is cooperating with the human end-user;

- *anthropomorphism:* the agent must be perceived to be specifically human (have a human personality, for example);

- *graceful degradation:* if communication with the end-user is not successful, then at least part of the task should be carried out properly, rather than having a total communication breakdown.

In this domain of investigation, the question has been put as to why we readily believe or become involved with animated cartoon characters, despite the fact that their physical representations are usually far from realistic. The suggested answer is that the perceived intentional states of the animated characters are much more compelling to the human observer than the degree of physical verisimilitude. Inspired by this phenomenon, attempts have been made to derive principles associated with intentions from the animated film industry and transfer these notions to computer systems interfaces so that users can more easily and comfortably interact with the systems (Bates, 1994).

While user interface agent developments are currently attracting considerable media attention, our research at the University of Otago is focused on software engineering agents and is concerned with the more general agents possessing the original five basic properties.

4 Software Engineering Agents Research at Otago

There is growing interest in agent-based software engineering support systems and tools that provide a higher-level environment for the construction of agent models and systems based on them. The motivation for this interest seems to be coming from two directions:

- The tendency to build larger, more distributed and reactive systems calls for improved modelling approaches that offer both a structural and behavioural view of the system. Software agents provide intuitively appealing modelling primitives (as outlined above) for this enterprise, and they can be combined with process modelling methods. The resulting tools and methods can not only assist *process modelling* for software development, but also facilitate the understanding of organisational processes and business process reengineering.

- With the great richness and diversity of today's computer information systems and the increasing degree to which they are embedded in the everyday processes of the world, there is a growing need to have these systems exchange information and services with each other. The resulting *interoperability* of these systems will enable the solutions of problems that could not be solved otherwise. Agent-based software systems are designed to facilitate this interoperability in a heterogeneous environment. Each software system or component is fashioned as an agent that communicates (negotiates) with other agents in order to operate effectively in its environment. They exchange data, logical scheme, and individual commands or programs, enabling them effectively to program each other in ways that are useful.

4.1 Process modelling with agent-based systems

There are two basic aspects to agent models that characterise such systems:

1. the nature or structure of the agent itself, and

2. the architecture of the system in which the agents interact.

The structure of the agent itself (item 1, above) can be implemented with object-oriented programming constructs, and the question arises as to what distinguishes agent-based software systems from object-oriented systems. The answer can be found in item 2. In order to communicate generally, the agents require an agent communication language that is independent of the specific structural features of individual agents, and it is this general agent communication language that represents a progressive step from ordinary object-oriented program message passing. At the moment there is work on several candidate agent communication language specifications (Finin *et al*., 1994; Genesereth and Ketchpel, 1994). In addition to the communication protocols among agents, item 2 relates to the manner in which agents are coordinated for interaction. One approach takes advantage of recent extensions in the area of Petri net theory (coloured Petri nets) (Jensen, 1990), and work based on this approach is being conducted at the University of Otago. With coloured Petri nets serving as the coordination framework and the individual coloured tokens of the net

serving as the agents, distributed information systems can be developed that have an intuitive graphical modelling representation and a formal representation for synchronisation and coordination among the individual elements. There has also been work in the area of representing the internal structure of causal agents themselves (not just the coordination among the agents) in terms of coloured Petri nets (Purvis, 1993; Levis, 1988). This latter effort could contribute to model refinement so that a computer representation of a system could be entirely in terms of Petri nets.

The most immediate modelling application for researchers in this area is the software construction process itself. Since evidence indicates that errors in software engineering projects are often introduced at the very earliest stages of development, *i.e.* during the requirements analysis stage, it is important that the facilities for initial software engineering models support a natural mapping from the real world to software structures. Traditional software engineering approaches, employing structured analysis, begin this mapping process by representing the static, structural relations among individual software elements in terms of entity-relationship diagrams (ERDs), and the behavioural elements of a system separately in terms of data flow diagrams (DFDs). Using the agent-based approach, one can replace the entities of ERDs with causal agents and the DFDs with coloured Petri nets, which results in a more integrated and intuitive modelling capability.

Since computer information systems are increasingly embedded in real-world operations and processes, it is important to be able to model the entire environment in which they operate. Agent-based process modelling is helpful in this regard, since it is a general modelling paradigm, and both humans and computer systems (as well as any other active entity) can be modelled as causal agents. In fact, agent-based process modelling can be used to model and simulate the behaviour of any organisation of interest. At the University of Otago, we are interested in modelling environmental resource management processes and are currently building a model of the dynamic processes associated with the national Resource Management Act of New Zealand in terms of agents and coloured Petri nets (Purvis and Purvis, 1995; Purvis *et al*., 1994).

4.2 Software interoperability

One way to increase software development productivity is to use a software engineering environment: an integrated collection of software and hardware tools tailored to support the team development of large, long-lifetime software systems (Sommerville, 1992). The major limitations of software engineering environments are their high cost and limited coverage of the software life-cycle. An easier solution would be to integrate all the separate tools that are currently being used into a single environment instead of attempting to develop or purchase one monolithic piece of software.

A common frustration when developing software is the need to convert the format of a document on one platform into a different format on another platform so it can be used by the appropriate tool in the next phase of the software life-cycle. This is because each tool has developed its own standard for files which others must conform to if they are to communicate. Even when a supposed standard has been created by the common use of a single language there are still problems (consider the problems that

commonly arise when printing PostScript files received from other people). The problem that has to be addressed is what to use as the communication protocol between components of an environment and what granularity the atoms of the protocol should have.

Tool integration can be seen at a number of levels of granularity: carrier level, lexical level, syntactic level, semantic level, and method level. With Unix, the character stream is the method of integration between components, and so each component must have encoded a method of converting the file format into a stream of characters and back again if components are to be placed in a pipeline. Thus any environment which used Unix pipes as the method of integration would duplicate a lot of code in the conversion process. While the character stream is the simplest method of tool integration, it leads to much duplicated effort; this effort is in the conversion of the communication format into terms understandable by the component. This level of cooperation is known as *carrier level integration* as the components simply exchange bytes.

Lexical level integration produces tightly coupled toolsets which share common data formats and operating conventions which allow them to interact in a meaningful way. The data format itself is embedded in the tools; adding a new tool means recoding the communication mechanisms for that tool.

Syntactic level integration is where tools cooperate via a predefined data structure and operations on that structure are available to all tools without the need for each tool to repeat the analysis, validation and conversion of the data structure. Examples of such integration are programming environments built around an underlying representation such as a parse tree.

Semantic level integration implies that a set of commonly agreed data structure definitions, and the meanings of the operations on those structures, is available. This is the level of integration currently possible with agent technology.

There is one further level of integration, namely *method level integration*. With this level, tools are used and interact only within the context of the software development process. It is not only necessary to agree on the data structures being manipulated and the manipulation, but also the development process within which the tools are being used. Numerous methods and methodologies exist supporting some part of the project life-cycle. It must be possible to combine methods to produce a consistent view of the life-cycle, and then to model that view in the environment itself and within the particular tools so that they "know" their role in the process and can interact with other tools by sending messages to let them know that they have just done something that they should be aware of and affects them.

Designing software tools to act as communicating, cooperative agents (or making existing tools act like agents by providing an agent-style software wrapper around them) would allow a set of tools to be integrated at the semantic level. To then achieve method level integration requires addressing three issues: which parts of the software life cycle are covered by each tool, in which order they need to be applied, and what parts of the software development process are not covered by the tools.

It was mentioned above that the semantic level of integration can be achieved with agents. This can be seen by viewing the problem of component communication in terms of knowledge transfer instead of data transfer. Instead of providing a standard file format that is necessarily a compromise, it would be better to provide a description of the data in terms every component understands. This is a problem of knowledge representation. What is needed is a standard knowledge representation language that every tool developer could use. This language would have to be expressive enough to encode the concepts of the domain of interest and extensible to allow new knowledge to be incorporated into the domain without invalidating existing components. Knowledge Interchange Formalism (KIF) (Genesereth and Ketchpel, 1994) is an example of such a language. It is a knowledge representation language based on first-order logic with some extensions. KIF allows a specification of the syntax and semantics of the problem domain to be produced and this then provides a standard way of communicating knowledge about the domain.

A software tool wishing to communicate or obtain some information must not only know how to encode that information; it must also determine the correct tools to communicate with and what protocols to use to locate and exchange information with them. To address these issues, Knowledge Query and Manipulation Language (KQML) (Genesereth and Ketchpel, 1994) has been proposed as a standard language for communication between software tools. KQML is a language that combines *performatives* (a term from speech act theory describing whether the message is an assertion, a query, a command, or other type of mutually agreed upon speech act) together with KIF statements built using words from the vocabulary of the domain. Two agents communicating must ensure that they have the same understanding of this vocabulary. This is the role of an *ontology*.

An ontology is an explicit specification of a conceptualization. A conceptualization is an abstract simplified view of the world that we wish to represent for some purpose. It is defined by the objects, concepts, and other entities that are presumed to exist in some area of interest, and the relationships that hold amongst them (Genesereth and Nilsson, 1987). Since the set of objects and relationships in an agent are reflected in the representational vocabulary, an ontology can be given as a set of definitions for this shared vocabulary (using some formal knowledge representation language such as KIF).

Another component of a KQML message is a specification of how the sender would like any reply to be delivered, *i.e.* which protocol should be followed. For example, a message representing a query about the price of a share of IBM stock might be encoded as:

```
(ask-one
    :content (PRICE IBM ?price)
    :sender   agent-99
    :receiver stock-server
    :language LPROLOG
    :ontology NYSE-TICKS)
```

In this example, the performative is *ask-one*, which will ask for a single reply; if we replace *ask-one* with *ask-all* we will get a set of replies.

KQML has implicit knowledge of communication protocols and supports dialogues between receivers and senders in a simple manner. Thus the resulting Agent Communication Language (ACL) comprises three parts: the ontology, an inner language (KIF) and an outer language (KQML).

CASE tool integration

CASE tool integration is a major problem (Brown and McDermid, 1991). As can be seen above there are many levels of integration that need to be examined. Many software communities have a number of CASE tools, written in several languages, running under different operating systems on different architectures. These tools will be communicating at various levels of integration, but usually only at the lexical or syntactic levels. The goal of CASE tool research at Otago is to integrate these tools into a software environment using method integration. The first stage is to produce an ontology for software engineering, or at least examine the toolsets available, abstract the knowledge they are attempting to capture, encode it in a knowledge representation language and use that as a basis of communication between the tools instead of using databases or files.

For most of these tools, a transducer must be created so that they logically communicate via the ontology but in reality are simply behaving as they were before; only the communication behaviour needs to be modelled. Where the source code is available a wrapper can be placed around the code to allow it to communicate directly via ACL. Once the existing code has become agent-aware (*i.e.* responds to agent messages) the tools can then be used by an agent that knows how to organise the software process; it will invoke an appropriate tool to carry out each portion of the software life-cycle. This can then allow a number of process models to be tested with existing tools. This implies that there must exist an ontology that deals with process models, and that an agent can advertise how much of the process it supports.

Desktop agents

As well as having applications to large scale, team software development, the agent-based approach to software interoperability has great potential for assisting end-users in their day-to-day work. The average computer site has many software packages, some bought "off the shelf" and other more specialised applications developed in-house or by external contractors. These will have been developed at various times and by different people in order to solve specific problems. In addition these applications may run under an assortment of different hardware platforms and operating systems. Often a user's work may involve taking the output from one or more programs, possibly applying some simple processing to it (such as changing the data format) and feeding the result to another program. The user could be much more productive if this process could be automated or at least eased in some way.

Standard data interchange formats and protocols such as OLE, CDIF and CORBA allow programs to share data more easily, but this is a low-level approach: there is no

way to represent the types of knowledge an application has or the services it provides. Creating agent 'wrappers' for the various software packages used so that they can communicate in a language such as KQML would allow the tasks that can be performed by each application to be represented declaratively. This knowledge could then be used by a special 'facilitator' agent to act as a central handler of the user's requests, farming out tasks to each application as appropriate (*e.g.* to query a database for particular information, to effect certain data transformations or to produce graphics from raw data).

As a testbed for research in this area it is proposed to take a particular end-user domain—the administration of a university programming course—and to create *desktop agents* that encapsulate the user's knowledge of how different software packages are used to support the overall administrative task. The packages that need to be integrated for this domain include the central university records system (running in-house software under VMS), a dBASE database containing student records for the course, a Visual Basic program for marking electronically submitted assignments on-line, a DOS program allowing students to view their marks on-line, Microsoft Excel (for preparing graphs and summaries of student marks) and various Unix, DOS and Novell Netware utilities. While most tasks performed using these tools are of a simple nature, transforming the data between the various formats required can be rather tedious. Integrating these tools using an agent-based approach should greatly decrease the amount of time spent on course administration and reduce the chance of errors introduced through manual data conversion.

5 Conclusion

The notion of intelligent agents that communicate in distributed heterogeneous environments is attracting attention not only as a useful approach for conceptualising complex systems but also as the basis for software implementations. As the computing world moves towards further integration of information systems across networks, this paradigm stands to become the focus of increasing research and software development. In this paper we have outlined some of the basic ideas behind agent-based computing and identified some of the distinctions between "software engineering agents" and "user interface agents". Also briefly discussed have been some of our research interests at the University of Otago concerning the use of software engineering agents in the areas of process modelling and software interoperability.

References

Bates, J. (1994). The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125.

Brown, A. W. and McDermid, J. A. (1991). On integration and reuse in a software development environment. In *Software Engineering Environments*, Vol. 3. Ellis Horwood.

Finin, T., Frizson, R., McKay, D., and McEntire, R. (1994). KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*. ACM Press.

Foner, L. N. (1993). What's an agent anyway? — a sociological case study. Report available by FTP, MIT Media Lab.

Genesereth, M. and Nilsson, N. (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann.

Genesereth, M. R. and Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, 37(7):48–53.

Guilfoyle, C. (1994). *Intelligent agents: the new revolution in software*. Ovum, London.

Hill, G. C. (1994). Cyber servants: Electronic 'agents' bring virtual shopping a bit closer to reality. *The Wall Street Journal*, 27 September 1994.

Jensen, K. (1990). *Coloured Petri Nets: a High Level Language for System Design and Analysis*. Springer-Verlag.

Levis, A. H. (1989). Human organisations as distributed intelligence systems. In Mladenov, D., editor, *Distributed Intelligence Systems, Methods and Applications: selected papers from the IFAC/IMACS Symposium, Varna, Bulgaria, 1988,* pages 13–18, Pergammon Press.

Malamud, C. (1995). Viewpoint: An airline-style price war looms in telecommunications. *IEEE Spectrum*, January 1995, page 32.

Nejmeh, B. A. (1994). Internet: A strategic tool for the software enterprise. *Communications of the ACM*, 37(11):23–27.

Ousterhout, J. (1995). Scripts and agents: The new software high ground. In *Proceedings of Winter USENIX Conference*, January 1995, New Orleans.

Purvis, M. K. (1993). Causal modelling in software engineering design. In Barta, B. Z., Hung, S. L., and Cox, K. R., editors, *Software Engineering Education*, pages 175–188. North Holland.

Purvis, M. K., Benwell, G. L., and Purvis, M. A. (1994). Dynamic modelling of the Resource Management Act. *New Zealand Journal of Computing*, 5(1):45–56.

Purvis, M. K. and Purvis, M. A. (1995). Modelling environmental legislative processes with Petri nets. In *Proceedings of the IASTED International Conference on Modelling and Simulation*, Pittsburgh.

Sargent, P. (1992). Back to school for a brand new ABC. *The Guardian*, 12 March 1992, page 28.

Shoham, Y. (1993). Agent-oriented programming. Artificial Intelligence, 60(1):51–92.

Sommerville, I. (1992). *Software Engineering*. Addison-Wesley, 4th edition.

White, J. E. (1994). Telescript technology: The foundation for the electronic marketplace. White paper, General Magic Inc., Mountain View, California.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, Vol. 10.