

1 Introduction

In past decades, software developers created massive, monolithic software programs that often performed a wide variety of tasks. During the past few years, however, there has been a shift from the development of massive programs containing millions of lines of code, to smaller, *modular*, pieces of code, where each module performs a well defined, focused task (or a small set of tasks), rather than thousands of different tasks, as used to be the case with old legacy systems. Software agents are the latest innovation in this trend towards splitting complex software systems into components. Roughly speaking, a software agent is a body of software that:

- provides one or more useful *services* that other agents may use under specified conditions,
- includes a *description* of the services offered by the software, which may be accessed and understood by other agents,
- includes the ability to *act autonomously* without requiring explicit direction from a human being,
- includes the ability to succinctly and declaratively describe how an agent determines *what actions to take* even though this description may be kept hidden from other agents,
- includes the ability to *interact* with other agents—including humans—either in a cooperative, or in an adversarial manner, as appropriate.

Note that not all software agents have to have the above properties—however any software agent programming paradigm must have the ability to create agents with some or all of these properties. In addition, agents will have a variety of other properties not covered in the above list, which will be spelled out in full technical detail as we go through this book.

With the proliferation of the Internet, there is now a huge body of data stored in a vast array of diverse, heterogeneous data sources, which is directly accessible to anyone with a network connection. This has led to the need for several agent based capabilities.

Data Integration Agents: Techniques to *mix and match, query, manipulate, and merge* such data together have gained increasing attention. Agents that can access heterogeneous data sources, and mix and match such data are increasingly important. Several agent based techniques for such data integration have been developed (Bayardo et al. 1997; Arens, Chee, Hsu, and Knoblock 1993; Brink, Marcus, and Subrahmanian 1995; Lu, Nerode, and Subrahmanian 1996; Chawathe et al. 1994).

Mobile Agents: The rapid evolution of the Java programming language (Horstmann and Cornell 1997) and the ability of Java applets to “move” across the network, executing byte-code at remote sites, has led to a new class of “mobile” agents (Rus, Gray, and Kotz 1997; Lande and Osjima 1998; Vigna 1998b; White 1997). If such agents are to autonomously form teams with other agents to cooperatively solve a problem, it is necessary that various techniques will be needed, such as techniques for describing agent services, for comprehending agent services, and for indexing and retrieving agent services, as well as techniques to facilitate interoperability between multiple agents.

Software Interoperability Agents: As the number of Java *applets* and other freely available and usable software deployed on the web increases, the ability to pipe data from one data source directly into one of these programs, and pipe the result into yet another program becomes more and more important. There is a growing body of research on agents that facilitate software interoperability (Patil, Fikes, Patel-Schneider, McKay, Finin, Gruber, and Neches 1997).

Personalized Visualization: Some years ago, the Internet was dominated by computer scientists. That situation has experienced a dramatic change and over the years, the vast majority of Internet users will view the Internet as a tool that supports their interests, which, in most cases, will not be computational. This brings with it a need for visualization and presentation of the results of a computation. As the results of a computation may depend upon the interests of a user, different visualization techniques may be needed to best present these results to the user (Candan, Prabhakaran, and Subrahmanian 1996; Ishizaki 1997).

Monitoring Interestingness: As the body of network accessible data gets ever larger, the need to identify what is of interest to users increases. Users do not want to obtain data that is “boring” or not relevant to their interests. Over the years, programs to monitor user interests have been built—for example, (Goldberg, Nichols, Oki, and Terry 1992; Foltz and Dumais 1992; Sta 1993; Sheth and Maes 1993) presents systems for monitoring newspaper articles, and several intelligent mail-handlers prioritize user’s email buffers. Techniques to identify user-dependent *interesting* data are growing increasingly important.

The above list merely provides a few simple examples of so-called *agent applications*. Yet, despite the growing interest in agents, and the growing deployment of programs that are billed as being “agents” several basic scientific questions have to be adequately answered.

(Q1) *What is an agent?*

Intuitively, any definition of agenthood is a predicate, `isagent`, that takes as input a program P in any programming language. Program P is considered an agent *if, by definition*,

$\text{isagent}(P)$ is true. Clearly, the isagent predicate may be defined in many different ways. For example, many of the proponents of Java believe that $\text{isagent}(P)$ is true *if and only if* P is a Java program—a definition that some might consider restrictive.

(Q2) *If program P is not considered to be an agent according to some specified definition of agenthood, is there a suite of tools that can help in “agentizing” P ?*

Intuitively, if a definition of agenthood is mandated by a standards body, then it is reasonable for the designer of a program P which does not comply with the definition of agenthood, to want tools that allow program P to be reconfigured as an agent. Efforts towards a definition of agenthood include ongoing agent standardization activities such as those of *FIPA* (the Foundation for Intelligent Physical Agents).

(Q3) *What kind of software infrastructure, is required for multiple agents to interact with one another once a specific definition of agenthood is chosen, and what kinds of basic services should such an infrastructure provide?*

For example, suppose agents are programs that have (among other things) an associated service description language in which each agent is required to describe its services. Then, yellow pages facilities which an agent might access are needed when the agent needs to find another agent that provides a service that it requires. Such a yellow pages service is an example of an infrastructural service.

The above questions allow a multiplicity of answers. For every possible definition of agenthood, we will require different agentization tools and infrastructural capabilities. The main aim of this book is to study what properties *any* definition of agenthood should satisfy. In the course of this, we will specifically make the following contributions.

- We will provide a concrete definition of agenthood that satisfies the requirements alluded to above, and compare this with alternative possible definitions of agenthood;
- We will provide an architecture and algorithms for agentizing programs that are deemed not to be agents according to the given definition;
- We will provide an architecture and algorithms for creating and deploying software agents that respect the above definition;
- We will provide a description of the infrastructural requirements needed to support such agents, and the algorithms that make this possible.

The rest of this chapter is organized as follows. We will first provide three motivating example applications in Sections 1.1, 1.2, and 1.3, respectively. These three examples will each illustrate different features required of agent infrastructures and different capabilities required of individual agents. Furthermore, these examples will be revisited over and over

throughout this entire book to illustrate basic concepts. In short, these examples form a common thread throughout this whole book. Later, in Section 1.4, we will provide a brief overview of existing research on software agents, and specify how these different existing paradigms address one or more of the basic questions raised by these three motivating examples. Section 1.4 will also explain what the shortcomings of these existing approaches are. In Section 1.5, we describe some general desiderata that agent theories and architectures should satisfy. Finally, in Section 1.6, we will provide a quick glimpse into the organization of this book, and provide a birdseye view of how (and where) the shortcomings pointed out in Section 1.4 are addressed by the framework described in the rest of this book.

1.1 A Personalized Department Store Application (STORE)

Let us consider the case of a large department store that has a web-based marketing site. Today, the Internet contains a whole host of such sites, offering on-line shopping services.

Today's Department Store: In most existing web sites today, interaction is initiated by a user who contacts the department store web site, and requests information on one or more consumer products he is interested in. For example, the user may ask for information on "leather shoes." The advanced systems deployed today access an underlying database and bring back relevant information on leather shoes. Such relevant information typically includes a picture of a shoe, a price, available colors and sizes, and perhaps a button that allows the user to place an order. The electronic department store of today is characterized by two properties: first, it assumes that users will come to the department store, and second, it does nothing more than simply retrieving data from a database and displaying it to the user.

Tomorrow's (Reactive) Department Store: In contrast, the department store of tomorrow will take explicit actions so that the department store goes to the customer, announcing items deemed to be of interest to the customer, rather than waiting for the customer to come to the store. This is because the department store's ultimate goal is to maximize profit (current as well as future), and in particular, it will accomplish this through the following means: It would like to ensure that a customer who visits it is presented items that maximize its expected profit as well as the likelihood of making a sale (e.g., they may not want to lose a sale by getting too greedy.) In particular, the department store would like to ensure that the items it presents a user (whether she visited the site of her own volition, or whether the presentation is a directed mailing), are items that are likely to be of maximal interest to the user—there is no point in mailing information about \$100-dollar ties to a person who has always bought clothing at lower prices.

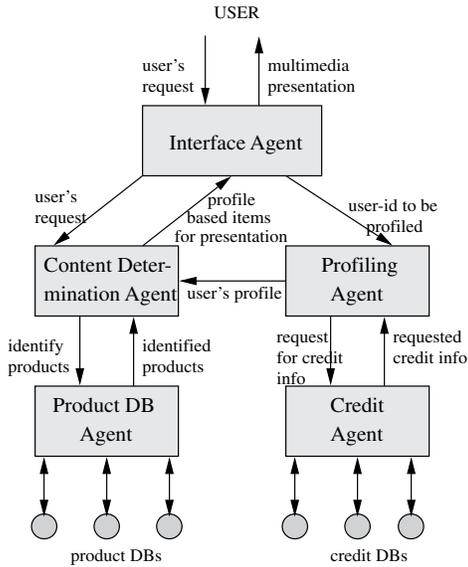


Figure 1.1
Interactions between Agents in STORE Example.

Intelligent agent technology may be used to accomplish these goals through a simple architecture, as shown in Figure 1.1. This architecture involves the following agents:

- 1. A Credit Database Agent:** This agent does nothing more sophisticated than providing access to a credit database. In the United States, many department stores issue their own credit cards, and as a consequence, they automatically have access to (at least some) credit data for many customers. The credit database agent may in fact access a variety of databases, not just one. Open source credit data is (unfortunately) readily available to paying customers.
- 2. Product Database Agent:** This agent provides access to one or more product databases reflecting the merchandise that the department store sells. Given a desired product description (e.g., “leather shoes”), this agent may be used to retrieve tuples associated with this product description. For example, a department store may carry 100 different types of leather shoes, and in this case, the product database may return a list of 100 records, one associated with each type of leather shoe.
- 3. A Profiling Agent:** This agent takes as input the identity of a user (who is interacting with the Department Store Interface agent described below). It then requests the credit database agent for information on this user’s credit history, and analyses the credit data.

Credit information typically contains detailed information about an individual's spending habits. The profiling agent may then classify the user as a "high" spender, an "average" spender, or a "low" spender. Of course, more detailed classifications are possible; it may classify the user as a "high" spender on clothing, but a "low" spender on appliances, indicating that the person cares more about personal appearance than on electrical appliances in his home.

As we go through this book, we will see that the Profiling agent can be made much more complex—if a user's credit history is relatively small (as would be the case with someone who pays cash for most purchases), it could well be that the Profiling agent analyzes other information (e.g., the person's home address) to determine his profile and/or it might contact other agents outside the department store that sell profiles of customers.

4. A Content Determination Agent: This agent tries to determine what to show the user. It takes as input, the user's request, and the classification of the agent as determined by the Profiling Agent. It executes a query to the product database agent, which provides it a set of tuples (e.g., the 100 different types of leather shoes). It then uses the user classification provided by the profiling agent to filter these 100 leather shoes. For example, if the user is classified as a "high spender," it may select the 10 most expensive leather shoes. In addition, the content determination agent may decide that when it presents these 10 leather shoes to the user, it will run advertisements on the bottom of the screen, showing other items that "fit" this user's high-spending profile.

5. Interface Agent: This agent takes the objects identified by the Content Determination Agent and weaves together a multimedia presentation (perhaps accompanied with music to the user's taste if it has information on music CDs previously purchased by the user!) containing these objects, together with any focused advertising information.

Thus far, we have presented how a department store might deploy a multiagent system. However, a human user may wish to have a personalized agent that finds an online store that provides a given service. For example, one of the authors was recently interested in finding wine distributors who sell *1990 Chateau Tayac* wines. An agent which found such a distributor would have been invaluable. In addition to finding a list of such distributors, the user might want to have these distributors ranked in descending order of the per bottle sales—the scenario can be made even more complex by wanting to have distributors ranked in descending order of the total (cost plus shipping) price for a dozen bottles.

Active Department Store of Tomorrow: Thus far, we have assumed that our department store agent is *reactive*. However, in reality, a department store system could be *proactive* in the following sense. As we all know, department stores regularly have sales. When a sale occurs, the department store could have a *Sale-Notification Agent* that performs the

following task. For every individual I in the department store's database, the department store could:

- identify the user's profile,
- determine which items going on sale "fit" the user's profile, and
- take an appropriate action—such an action could email the user a list of items "fitting" his profile. Alternatively, the action may be to create a *personalized sale flyer* specifying for each user, a set of sale item descriptions to be physically mailed to him.

In addition, the Sale-Notification agent may *schedule future actions* based on its *uncertain beliefs about the users*. For example, statistical analysis of John Doe's shopping habits at the store may indicate the following distribution:

Day	Percentage Spent
Monday	2%
Tuesday	3%
Wednesday	3%
Thursday	2%
Friday	27%
Saturday	50%
Sunday	13%

In the above table, the tuple, $\langle \text{Monday}, 2\% \rangle$ means that of all the money that John Doe is known to have spent at this store, 2% of the money was spent on Mondays.

The Sale-Notification agent may now reason as follows: 90% of John Doe's dollars spent at this store are spent during the Friday-Saturday-Sunday period. Therefore, I will mail John Doe promotional material on sales so as to reach him on Thursday evening.

However, there may be uncertainty in postal services. For example, the bulk mailing system provided by the US Postal Service may have statistical data showing that 13% of such mailings reach the customer within 1 day of shipping, 79% in 2 days, and the remaining 8% take over 2 days. Thus, the Sale-Notification agent may mail the sales brochures to John Doe on Tuesday.

When we examine the above department store example, we notice that:

1. The department store example may be viewed as a multiagent system where the interactions between the agents involved are clear and well defined.
2. Each agent has an associated body of data structures and algorithms that it maintains. The content of these data structures may be updated independently of the application as

a whole (e.g., user's credit data may change in the above example without affecting the Product-Database agent).

3. Each agent is capable of performing a small, but well defined set of actions/tasks.
4. The actual actions executed (from the set of actions an agent is capable of performing) may vary depending upon the circumstances involved. For example, the Credit agent may provide credit information in the above example only to the Profiling Agent, but may refuse to respond to credit requests from other agents.
5. Each agent may reason with beliefs about the behavior of other agents, and each agent not only decides what actions to perform, but also when to perform them. Uncertainty may be present in the beliefs the agent holds about other agents.

1.2 The Controlled Flight into Terrain Application (CFIT)

According to the *Washington Post* (Feb. 12, 1998, page A-11) 2,708 out of 7,496 airline fatalities during the 1987–1996 period did not happen due to pilot error (as is commonly suspected), but due to a phenomenon called *controlled flight into terrain* (CFIT). Intuitively, a CFIT error occurs when a plane is proceeding along an Auto-Pilot (not human) controlled trajectory, but literally crashes into the ground. CFIT errors occur because of malfunctioning sensors and because the autopilot program has an *incorrect belief* about the actual location of the plane. CFIT is the number one cause of airline deaths in the world. The CFIT problem is highlighted by two major plane crashes during recent years:

- The December 1995 crash of an American Airlines plane in Cali, Colombia, killing over 250 people including Paris Kanellakis, a prominent computer scientist;
- the crash of a US military plane near Dubrovnik, Yugoslavia in 1996, killing the US Commerce Secretary, Ron Brown.

We have developed a preliminary solution to the CFIT problem, and have developed a working prototype of a multi-agent solution to the CFIT problem. The solution involves the following agents:

Auto-Pilot Agent: The Auto-Pilot agent ensures that the plane stays on its allocated flight path. Most civilian flights in the world fly along certain prescribed flight corridors that are assigned to each flight by air traffic controllers. The task of the Auto-Pilot agent is to ensure that the plane stays on-course, and make appropriate adjustments (by perhaps using AI planning or 3-dimensional path planning techniques) when the physical dynamics of the plane cause it to veer off course. Techniques for agent based solutions to flight planning and

air traffic control problems have been studied in the agents community by Tambe, Johnson, and Shen (1997).

Satellite Agents: We assume the existence of a set of satellite agents that will monitor the position of several planes simultaneously. Every Δt units of time, each satellite agent broadcasts a report that may be read by the location agent. Thus, if $\Delta t = 10$ and the first report is read at time 0, then this means that all the satellite agents send reports at times 0, 10, 20, . . . and so on. Each satellite agent specifies where it believes the plane is at that point in time.

GPS Agent: This agent takes reports from multiple satellite agents above and merges them together. Multiplexing satellite agents together enhances reliability—if one satellite agent fails, the others will still provide a report. Merging techniques may include methods of eliminating outliers—e.g., if 9 of 10 satellite agents tell the plane it is at location *A* and the 10th agent tells the plane it is at location *B*, the last report can be eliminated. The GPS agent then feeds the GPS-based location of the plane to the Auto-Pilot agent, which consults the Terrain agent below before taking corrective action.

Terrain Agent: The Terrain agent takes a coordinate in the globe, and retrieves a terrain map for the region. In the case of our CFIT example, a special kind of terrain map is retrieved called a *Digital Terrain Elevation Data (DTED)* map. Our implementation currently includes DTED data for the whole of the continental USA, but not for the world. Given any (x, y) location which falls within this map, the elevation of that (x, y) location can then be retrieved from the DTED map by the Terrain agent. The Terrain agent provides to the Auto-Pilot agent a set of “no-go” areas. Using this set, the Auto-Pilot agent can check if its current heading will cause it to fly into a mountain (as happened with the American Airlines crash of 1996), and in such cases, it can replan to ensure that the plane avoids these no-go areas.

Figure 1.2 on the next page shows a schematic diagram of the different agents involved in this example.

The reader will readily note that there are some similarities, as well as some differences, between this CFIT example and the preceding STORE example. The example is similar to the department store example in the following ways:

- Like the STORE application, the CFIT application may be viewed as a multiagent system where the agents interact with one another in clearly defined ways.
- In both examples, each agent manages a well defined body of data structures and associated algorithms, but these data structures may be updated autonomously and vary from one agent to another.
- As in the case of the STORE example, each agent performs a set of well defined tasks.

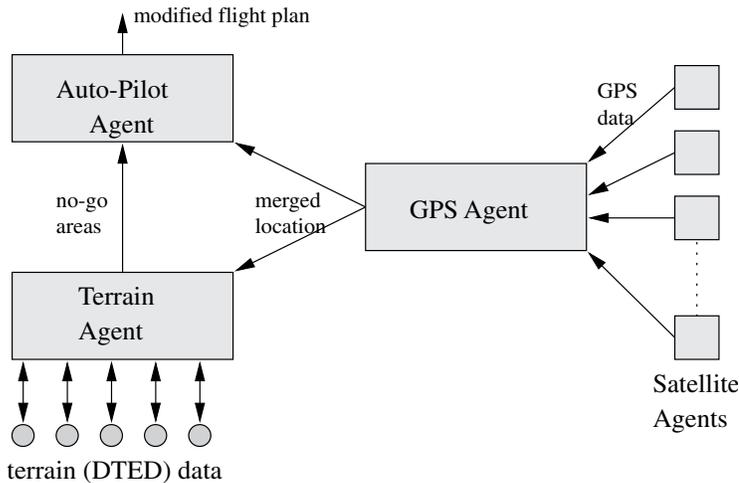


Figure 1.2
Interactions between Agents in CFIT Example.

- As in the case of the STORE example, agents may take different actions, based on the circumstances. For example, some satellite agents may send updates to one plane every 5 seconds, but only at every 50 seconds for another plane.

In addition, the following attributes (which also appear in the department store example) play an important role in the CFIT example:

Reasoning about Beliefs: The Auto-Pilot agent reasons with *Beliefs*. At any given point t in time, the Auto-Pilot agent believes that it is at a given location ℓ_t . However, its belief about its location, and the location it is really at, may be different. The task of the GPS agent in the CFIT application is to alert the Auto-Pilot agent to its incorrect beliefs, which may then be appropriately corrected by the Auto-Pilot agent.

In this example, the Auto-Pilot agent believes the correction it receives from the satellite agents. However, it is conceivable that if our plane is a military aircraft, then an enemy might attempt to masquerade as a legitimate satellite agent, and falsely inform the Auto-Pilot agent that it is at location ℓ_t^* , with the express intent of making the plane go off-course. However, agents must make decisions on how to act when requests/information are received from other agents. It is important to note that which actions an agent decides to execute depends upon background information that the agent has. Thus, if an agent suspects that a satellite agent message is not reliable, then it might choose to ignore information it receives from that agent

or it may choose to seek clarification from another source. On the other hand, if it believes that the satellite agent's message is "legitimate," then it may take the information provided into consideration when making decisions. In general, agents decide how to act, based upon (i) the background knowledge that the agent has, and (ii) the beliefs that the agent currently holds.

Delayed Actions: Yet another difference with the STORE example is that the Auto-Pilot agent may choose to *delay* taking actions. In other words, the Auto-Pilot agent may know at time t that it is off-course. It could choose to create a plan at time t (creation of a plan is an explicit action) that commits the Auto-Pilot agent to take other actions at later points in time, e.g., "Execute a climb action by 50 feet per second between time $(t + 5)$ and time $(t + 10)$."

Uncertainty: If the Auto-Pilot agent receives frequent information from the Location agent, stating that it is off-course, it might suspect that some of its on-board sensors or actuators are malfunctioning. Depending upon its knowledge of these sensors and actuators, it might have different beliefs about which sensor/actuator is malfunctioning. This belief may be accompanied with a *probability* or *certainty* that the belief is in fact true. Based on these certainties, the Auto-Pilot may take one of several actions that could include returning the plane to manual control, switching off a sensor and/or switching on an alternative sensor. In general, in extended versions of our CFIT example, Auto-Pilot agents may need to reason with uncertainty when making decisions.

1.3 A Supply Chain Example (CHAIN)

Supply chain management (Bowersox, Closs, and Helfferich 1986) is one of the most important activities in any major production company. Most such companies like to keep their production lines busy and on schedule. To ensure this, they must constantly monitor their inventory to ensure that components and items needed for creating their products are available in adequate numbers.

For instance, an automobile company is likely to want to guarantee that they always have an adequate number of tires and spark plugs in their local inventory. When the supply of tires or spark plugs drops to a certain predetermined level, the company in question must ensure that new supplies are promptly ordered. This may be done through the following steps.

- In most large corporations, the company has "standing" contracts with producers of different parts (also referred to as an "open" purchase order). When a shortfall occurs, the company contacts suppliers to see which of them can supply the desired quantity of the item(s) in question within the desired time frame. Based on the responses received from the suppliers, one or more purchase orders may be generated.

- The company may also have an existing purchase order with a large transportation provider, or with a group of providers. The company may then choose to determine whether the items ordered should be: (a) delivered entirely by truck, or (b) delivered by a combination of truck and airplane.

This scenario can be made significantly more sophisticated than the above description. For example, the company may request bids from multiple potential suppliers, the company may use methods to identify alternative substitute parts if the ones being ordered are not available, etc. For pedagogical purposes, we have chosen to keep the scenario relatively simple.

The above automated purchasing procedure may be facilitated by using an architecture such as that shown in Figure 1.3. In this architecture, we have an Inventory agent that

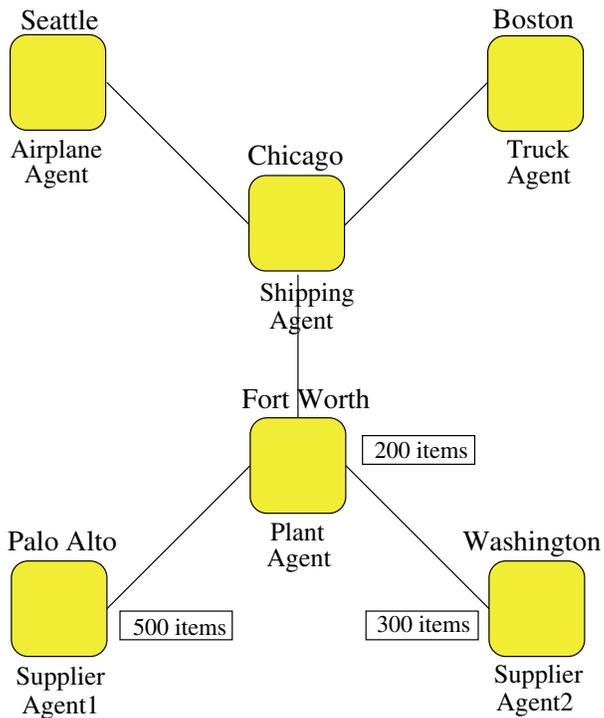


Figure 1.3
Agents in CHAIN Example.

monitors the available inventory at the company's manufacturing plant. We have shown two suppliers, each of which has an associated agent that monitors two databases:

- An ACCESS database specifying how much uncommitted stock the supplier has. For example, if the tuple $\langle \text{widget50}, 9000 \rangle$ is in this relation, then this means that the supplier has 9000 pieces of widget50 that haven't yet been committed to a consumer.
- An ACCESS database specifying how much committed stock the supplier has. For example, if the triple $\langle \text{widget50}, 1000, \text{companyA} \rangle$ is in the relation, this means that the supplier has 1000 pieces of widget50 that have been committed to company A.

Thus, if company-B were to request 2000 pieces of widget50, we would update the first relation, by replacing the tuple $\langle \text{widget50}, 9000 \rangle$ by the tuple $\langle \text{widget50}, 7000 \rangle$ and adding the tuple $\langle \text{widget50}, 2000, \text{companyB} \rangle$ to the latter relation—assuming that company B did not already have widget50 on order.

Once the Plant agent places orders with the suppliers, it must ensure that the transportation vendors can deliver the items to the company's location. For this, it consults a *Shipping-Agent*, which in turn consults a *Truck-Agent* (that provides and manages truck schedules using routing algorithms) and an *Airplane-Agent* (that provides and manages airplane freight cargo). The truck agent may in fact control a set of other agents, one located on each truck. The truck agent we have built is constructed by building on top of ESRI's MapObject system for route mapping. These databases can be made more realistic by adding other fields—again for the sake of simplicity, we have chosen not to do so.

As in the previous two examples, the Plant agent may make decisions based on a more sophisticated reasoning process. For example:

Reasoning about Uncertainty: The Plant agent may have some historical data about the ability of the supplier agents to deliver on time. For example, it may have a table of the form:

Supplier	Item	Days Late	Percentage
supplier1	widget1	-3	5
		-1	10
		0	55
		1	20
		2	10
...
...

In this table, the first tuple says that in cases where `supplier1` promised to deliver `widget1`, he supplied it 3 days early in 5% of the cases. The last entry above likewise says that when `supplier1` promised to deliver `widget1`, he supplied it 2 days late in 10% of the cases. Using this table, the Plant agent may make decisions about the probability that placing an order with `supplier1` will in fact result in the order being delivered within the desired deadline.

Delayed Actions: When placing an order with `supplier1`, the Plant agent `plant` may want to retain the option of cutting the contract to `supplier1` if adequate progress has not been made. Thus, the Plant agent may inform `supplier1` up front that 10 days after placement of the order, it will inspect the status of the supplier's performance on that order (such inspections will of course be based on reasonable and precisely stated evaluation conditions). If the performance does not meet certain conditions, it might cancel part of the contract.

Reasoning about Beliefs: As in the case of the CFIT agent, the Plant agent may make decisions based on its *beliefs* about the suppliers ability to deliver, or the transportation companies ability to ship products. For example, if the Plant agent believes that a Transportation agent is likely to have a strike, it might choose to place its transportation order with another company.

The CHAIN example, like the other examples, may be viewed as a multiagent system where the interactions between agents are clearly specified, and each agent manages a set of data structures that can be autonomously updated by the agent. Furthermore, different agents may manage different data structures.

However, a distinct difference occurs when the Plant agent realizes that neither of its Supplier agents can supply the item that is required within the given time frame. In such a case, the Plant agent may need to *dynamically find* another agent that supplies the desired item. This requires that the Plant agent has access to some kind of yellow pages facility that keeps track of the services offered by different agents. Later, in Chapters 2 and 3, we will define detailed yellow pages service mechanisms to support the need of finding agents that provide a service, when the identity of such agents is not known *a priori*.

1.4 Brief Overview of Related Research on Agents

In this section, we provide a brief overview of existing work on agents, and explain their advantages and disadvantages with respect to the three motivating examples introduced above.

As we have already observed, the three examples above all share a common structure:

- Each agent has an associated set of data structures.
- Each agent has an associated set of low-level operations to manipulate those data structures.
- Each agent has an associated set of high-level actions that “weave” together the low level operations above that it performs.
- Each agent has a policy that it uses to determine which of its associated high level actions to execute in response to requests and/or events (e.g., receipt of data from another agent).

There are various other parameters associated with any single agent that we will discuss in greater detail in later chapters, but for now, these are the most salient features of practical implemented agents. In addition, a platform to support multi-agent interactions must provide a set of common services including, but not limited to:

1. *Registration* services, through which an agent can register the services it provides.
2. *Yellow pages* services that allow an agent to find another agent offering a service *similar* to a service sought by the agent.
3. *Thesauri* and *dictionary* services that allow agents to determine what words mean.
4. More sophisticated *ontological* services that allow an agent to determine what another agent might mean when it uses a term or expression.
5. *Security* services that allow an agent to look up the security classification of another agent (perhaps under some restricted conditions).

Different parts of various technical problems raised by the need to create multiagent systems have been addressed in many different scientific communities, ranging from the database community, the AI community, the distributed objects community, and the programming languages community, to name a few. In this section, we will briefly skim some of the major approaches to these technical problems—a detailed and much more comprehensive overview is contained in Chapter 13.

1.4.1 Heterogeneous Data/Software Integration

One of the important aspects of agent systems is the ability to uniformly access heterogeneous data sources. In particular, if agent decision making is based on the content of arbitrary data structures managed by the agent, then there must be some unified way of accessing

those data structures. Many formalisms have been proposed to integrate heterogeneous data structures. These formalisms fall into three categories:

Logical Languages: One of the first logical languages to integrate heterogeneous data sources was the *SIMS* system (Arens, Chee, Hsu, and Knoblock 1993) at USC which uses a LISP-like syntax to integrate multiple databases as well. More or less at the same time as *SIMS*, a Datalog-extension to access heterogeneous data sources was proposed in the *HERMES* Heterogeneous Reasoning and Mediator System Project in June 1993 (Lu, Nerode, and Subrahmanian 1996; Subrahmanian 1994; Brink, Marcus, and Subrahmanian 1995; Marcus and Subrahmanian 1996; Adali, Candan, Papakonstantinou, and Subrahmanian 1996; Lu, Moerkotte, Schue, and Subrahmanian 1995). Shortly thereafter, the IBM-Stanford *TSIMMIS* effort (Chawathe et al. 1994) proposed logical extensions of Datalog as well. These approaches differed in their expressive power—for instance, *TSIMMIS* was largely successful on relational databases, but also accessed some non-relational data sources such as bibliographic data. *SIMS* accessed a wide variety of AI knowledge representation schemes, as well as traditional relational databases. In contrast, *HERMES* integrated arbitrary software packages such as an Army Terrain Route Planning System, Jim Hendler’s UM Nonlin nonlinear planning system, a face recognition system, a video reasoning system, and various mathematical programming software packages.

SQL Extensions: SQL has long had a mechanism to make “foreign function” calls whereby an SQL query can embed a subquery to an external data source. The problem with most existing implementations of SQL is that even though they can access these external data sources, they make assumptions on the format of the outputs returned by such foreign function calls. Thus, if the foreign functions return answers that are not within certain prescribed formats, then they cannot be processed by standard SQL interpreters. Extensions of SQL to access heterogeneous relational databases such as the Object Database Connectivity (ODBC) standard (Creamer, Stegman, and Signore 1995) have received wide acceptance in industry.

OQL Extensions: Under the aegis of the US Department of Defense, a standard for data integration was proposed by a group of approximately 11 researchers selected by *DARPA* (including the first author of this book). The standard is well summarized in the report of this working group (Buneman, Ullman, Raschid, Abiteboul, Levy, Maier, Qian, Ramakrishnan, Subrahmanian, Tannen, and Zdonik 1996). The approach advocated by the *DARPA* working group was to build a *minimal core language* based on the Object Definition Language and the Object Query Language put forth earlier by the industry wide Object Data Management Group (*ODMG*) (Cattell et al. 1997). The basic idea was that the core part be a restricted version of OQL, and all extensions to the core would handle complex data types with methods.

Another important later direction on mediation includes the *InfoSleuth* effort (Bayardo et al. 1997) system, at MCC—this will be discussed in detail later in Chapter 4.

Implementations of all the three frameworks listed above were completed in the 1993–1996 time frame, and many of these are available, either free of charge or for a licensing fee (Brink, Marcus, and Subrahmanian 1995; Adali, Candan, Papakonstantinou, and Subrahmanian 1996; Lu, Nerode, and Subrahmanian 1996; Chawathe et al. 1994; Arens, Chee, Hsu, and Knoblock 1993). *Any of the frameworks listed above could constitute a valid language, by using which access is provided to arbitrary data structures.*

1.4.2 Agent Decision Making

There has been a significant amount of work on agent decision making. Rosenschein (1985) was perhaps the first to say that agents act according to states, and which actions they take are determined by rules of the form “*When P is true of the state of the environment, then the agent should take action A.*” Rosenschein and Kaelbling (1995) extend this framework to provide a basis for such actions in terms of situated automata theory. For example, in the case of the department store example, the Profiling Agent may use a rule of the form “*If the credit data on person P shows that she spends over \$200 per month (on the average) at our store, then classify P as a high spender.*” Using this rule, the Sales agent may take another action of the form “*If the Profiling agent classifies person P as a high spender, then send P material M by email.*”

Bratman, Israel, and Pollack (1988) define the *IRMA* system which uses similar ideas to generate plans. In their framework, different possible courses of actions (Plans) are generated, based on the agent’s intentions. These plans are then evaluated to determine which ones are consistent and optimal with respect to achieving these intentions. This is useful when applied to agents which have intentions that might require planning (though there might be agents that do not have any intentions or plans such as a GPS receiver in the CFIT example). Certainly, the Auto-Pilot agent in the CFIT example has an *intention*—namely to stay on course, as specified by the flight plan filed by the plane, and it may need to *replan* when it is notified by the GPS agent that it has veered off course.

The *Procedural Reasoning System (PRS)* is one of the best known multiagent construction systems that implements BDI agents (BDI stands for *Belief, Desires, Intentionality*) (d’Inverno, Kinny, Luck, and Wooldridge 1997). This framework has led to several interesting applications including a practical, deployed application called *OASIS* for air traffic control in Sydney, Australia. The theory of *PRS* is captured through a logic based development, in Rao and Georgeff (1991).

Singh (1997) is concerned about heterogeneity in agents, and he develops a theory of agent interactions through workflow diagrams. Intuitively, in this framework, an agent is

viewed as a finite state automaton. Agent states are viewed as states of the automaton, and agent actions are viewed as transitions on these states. This is certainly consistent with the three motivating examples—for instance, in the CHAIN example, when the Supplier1 agent executes an action (such as shipping supplies), this may certainly be viewed as a state transition, causing the available quantity of the supply item in question at Supplier1's location to drop.

1.4.3 Specific Interaction Mechanisms for Multiagent Systems

There has been extensive work in AI on specific protocols for multiagent interactions. Two such mechanisms are worth mentioning here:

Bidding Mechanisms: Let us return to the CHAIN example and assume that neither of the two approved suppliers (with existing contracts to funnel the purchase through) can deliver the supplies required by the Plant agent. In this case, the Plant agent needs to find another agent (one for which no contract is currently in force). The Plant agent needs to *negotiate* with the new agent, arriving at a mutually agreeable arrangement. There has been extensive work on negotiation in multiagent systems, based on the initial idea of contract nets, due to Smith and Davis (1983). In this paradigm, an agent seeking a service invites bids from other agents, and selects the bid that most closely matches its own. Schwartz and Kraus (1997) present a model of agent decision making where one agent invites bids (this is an action !) and others evaluate the bids (another action) and respond. Other forms of negotiation have also been studied and will be discussed in detail in Chapter 14.

Coalition Formation: A second kind of interaction between agents is coalition formation. Consider an expanded version of the CFIT example, in a military setting. Here, a Tank agent tank may have a mission, but as it proceeds toward execution of the mission, it encounters heavier resistance than expected. In this case, it may dynamically team with a helicopter gunship whose Auto-Pilot and control mechanisms are implemented using the CFIT example. Here, the tank is forming a *coalition dynamically* in order to accomplish a given goal. Coalition formation mechanisms where agents dynamically team up with other agents has been intensely studied by many researchers (Shehory, Sycara, and Jha 1997; Sandholm and Lesser 1995; Wooldridge and Jennings 1997). Determining which agents to team with is a sort of decision making capability.

1.4.4 Agent Programming

Shoham (1993) was perhaps the first to propose an explicit programming language for agents, based on object oriented concepts, and based on the concept of an agent state. In Shoham's approach, an agent

“is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments.”

He proposes a language, **Agent-0**, for agent programming, that provides a mechanism to express actions, time, and obligations. **Agent-0** is a simple, yet powerful language.

Closely related to Shoham’s work is that of Hindriks, de Boer, van der Hoek, and Meyer (1997) where an agent programming language based on BDI-agents is presented. They proceed upon the assumption that an agent language must have the ability to update its beliefs and its goals, and it must have a practical reasoning method (which will find a way to achieve goals). Hindriks, de Boer, van der Hoek, and Meyer (1997, p. 211) argue that “*Now, to program an agent is to specify its initial mental state, the semantics of the basic actions the agent can perform, and to write a set of practical reasoning rules.*”

When compared to Singh’s approach described earlier in this chapter, these approaches provide a compact way of representing a massive finite state automaton (only the initial state is explicit) and transitions are specified through actions and rules governing the actions. This is very appealing, and the semantics is very clean.

However, both approaches assume that all the reasoning done by agents is implemented in one form of logic or another, and that all agents involved manipulate logical data. While logic is a reasonable *abstraction* of data, it remains a fact of life that the vast majority of data available today is in the form of non-logical data structures that vary widely.

The second assumption made is that *all* reasoning done by agents is encoded through logical rules. While this is also reasonable as an abstraction, it is rarely true in practice. For example, consider the planning performed by the Auto-Pilot agent in the **CFIT** example, or the profiling performed by the Profiling agent in the department store example, or the route planning performed by the Truck agent in the in the **CHAIN** example. These three activities will, in all likelihood, be programmed using imperative code, and mechanisms such as those alluded to above must be able to meaningfully reason on top of such legacy code.

1.4.5 Agent Architectures

An architecture for the creation and deployment of multiagent applications must satisfy three goals:

1. First and foremost, it must provide an architecture for designing software agents.
2. It must provide the underlying software infrastructure that provides a common set of services that agents will need.
3. It must provide mechanisms for interactions between clients and the underlying agent infrastructure.

As we have already discussed the first point earlier in this section, we will confine ourselves to related work on the latter two components.

With respect to agent architectures, there have been numerous proposals in the literature, e.g., (Gasser and Ishida 1991; Glicoe, Staats, and Huhns 1995; Birmingham, Durfee, Mullen, and Wellman 1995), which have been broadly classified by Genesereth and Ketchpel (1994) into four categories:

1. In the first category, each agent has an associated “transducer” that converts all incoming messages and requests into a form that is intelligible to the agent. In the context of our CFIT example introduced this means that each agent in the example must have the ability to understand messages sent to it by other agents. However, the CFIT example shows only a small microcosm of the functioning of the Auto-Pilot agent. In reality, the Auto-Pilot needs to interact with agents associated with hundreds of sensors and actuators, and to require that the transducers anticipate what other agents will send and translate it is clearly a complex problem. In general, in an n -agent system, we may need $O(n^2)$ transducers, which is clearly not desirable.
2. The second approach is based on wrappers which “*inject code into a program to allow it to communicate*” (Genesereth and Ketchpel 1994, p. 51). This idea is based on the principle that each agent has an associated body of code that is expressed in a common language used by other agents (or is expressed in one of a very small number of such languages). This means that in the case of the CFIT example, each agent is built around a body of software code, and this software code has an associated body of program code (expressed perhaps in a different language) expressing some information about the program.
3. The third approach described in (Genesereth and Ketchpel 1994) is to completely rewrite the code implementing an agent, which is obviously a very expensive alternative.
4. Last but not least, there is the *mediation* approach proposed by Wiederhold (1993), which assumes that all agents will communicate with a mediator which in turn may send messages to other agents. The mediation approach has been extensively studied (Arens, Chee, Hsu, and Knoblock 1993; Brink, Marcus, and Subrahmanian 1995; Chawathe et al. 1994; Bayardo et al. 1997). However, it suffers from a problem. Suppose all communications in the CFIT example had to go through such a mediator. Then if the mediator malfunctions or “goes down,” the system as a whole is liable to collapse, leaving the plane in a precarious position. In an agent based system, we should allow point to point communication between agents without having to go through a mediator. This increases reliability of the entire multiagent system as a whole and often avoids inefficiency by avoiding huge workloads on certain agents or servers or network nodes.

1.4.6 Match-making Services

As stated before, one of the infrastructural tasks to be provided is yellow pages services whereby agents may advertise services they offer (via the yellow pages) and the infrastructure layer allows for identifying agents A that provide a service *similar* to a service requested by agent B . For instance, in the CHAIN example, the plant agent may need to contact such a yellow pages service in order to find agents that can provide the supply item needed. The yellow pages agent must attempt to identify agents that provide either the exact supply item required, or something similar to the requested item. Kuokka and Harada (1996) present the *SHADE* and *COINS* systems for matchmaking. *SHADE* uses logical rules to support matchmaking—the logic used is a subset of *KIF* and is very expressive. In contrast, *COINS* assumes that a message is a document (represented by a weighted term vector) and retrieves the *most similar* advertised services using the *SMART* algorithm of Salton and McGill (1983). Decker, Sycara, and Williamson (1997) present matchmakers that store capability advertisements of different agents. They look for *exact* matches between requested services and retrieved services, and concentrate their efforts on architectures that support load balancing and protection of privacy of different agents.

1.5 Ten Desiderata for an Agent Infrastructure

In this book, we will describe advances in the construction of agents, as well as multiagent systems. Our intent is to provide a rich formal theory of agent construction and agent interaction that is *practically* implementable and realizable. *IMPACT (Interactive Maryland Platform for Agents Collaborating Together)* is a software platform for the creation and deployment of agents, and agent based systems. In this book, we will provide one set of answers to the following questions raised at the beginning of this chapter:

- (Q1) *What is an agent?*
- (Q2) *If program P is not considered to be an agent according to some specified definition of agenthood, is there a suite of tools that can help in “agentizing” P ?*
- (Q3) *Once a specific definition of agenthood is chosen, what kind of software infrastructure is required to support interactions between such agents, and what core set of services must be provided by such an infrastructure?*

In particular, any solution to the above questions must (to our mind) satisfy the following important desiderata:

- (D1) Agents are for everyone: anybody who has a software program P , either custom designed to be an agent, or an existing legacy program, must be able to *agentize* their program

and *plug* it into the provided solution. In particular, in the case of the CFIT example, this means that if a new Satellite agent becomes available, or a better flight planning Auto-Pilot agent is designed, plugging it in should be simple. Similarly, if a new Supplier agent is identified in the CHAIN example, we should be able to access it easily and incorporate it into the existing multiagent CHAIN example. Any theory of agents must encompass the above diversity.

(D2) No theory of agents is likely to be of much practical value if it does not recognize the fact that data is stored in a wide variety of data structures, and data is manipulated by an existing corpus of algorithms. If this is not taken into account in a theory of agents, then that theory is not likely to be particularly useful.

(D3) A theory of agents must *not* depend upon the set of actions that the agent performs. Rather, the set of actions that the agent performs must be a *parameter* that is taken into account in the semantics. Furthermore, any proposed action framework must allow actions to have effects on arbitrary agent data structures, and must be capable of being built seamlessly on top of such existing applications.

(D4) Every agent should execute actions based on some *clearly articulated* decision policy. While this policy need not be disclosed to other agents, such a specification is invaluable when the agent is later modified. We will argue that a *declarative* framework for articulating decision policies of agents is imperative.

(D5) Any agent construction framework must allow agents to perform the following types of reasoning:

- Reasoning about its beliefs about other agents.
- Reasoning about uncertainty in its beliefs about the world and about its beliefs about other agents.
- Reasoning about time.

These capabilities should be viewed as *extensions* to a core agent action language, that may be “switched” on or off, depending upon the reasoning needs of an agent. The reason for this is that different agents need to reason at different levels of sophistication. However, increasingly sophisticated reasoning comes at a computational price, viz. an increase in complexity (as we will see in the book).

Thus, it is wise to have a *base language* together with a *hierarchy* of extensions of the base language reflecting increased expressive power. Depending on which language within this hierarchy the agent wishes to use, the computational price to be paid by the agent should be clearly defined. This also requires that we have a *hierarchy* of compilers/interpreters mirroring the language hierarchy. It is in general computationally unwise to use a solver for a language “high” in the hierarchy if an agent is using a language “low” in the hierarchy (e.g.,

using a solver for a PSPACE-complete problem on a polynomial instance of the problem is usually not wise).

(D6) Any infrastructure to support multiagent interactions *must* provide two important types of security—security on the agent side, to ensure that an agent (if it wishes) can protect some of its information and services, and security on the infrastructural side so that one agent cannot masquerade as another, thus acquiring access to data/services that is not authorized to receive.

(D7) While the efficiency of the code underlying a software agent cannot be guaranteed (as it will vary from one application to another), guarantees are needed that provide information on the performance of an agent relative to an oracle that supports calls to underlying software code. Such guarantees must come in two forms—results on *worst case complexity* as well as accompanying *experimental results*. Both these types of results are useful, because in many cases, worst case complexity results do not take into account specific patterns of data requests that become apparent only after running experiments. Conversely, using experimental results alone is not adequate, because in many cases, we do want to know worst case running times, and experimental data may hide such information.

(D8) Efficiency of an implementation of the theory is critical in the development of a multiagent system. We must identify efficiently computable *fragments* of the general hierarchy of languages alluded to above, and our implementations must take advantage of the specific structure of such language fragments. A system built in this way must be accompanied by a suite of software tools that helps the developer build sophisticated multiagent systems.

(D9) A critical point is *reliability*—there is no point in a highly efficient implementation, if all agents deployed in the implementation come to a grinding halt when the agent “infrastructure” crashes.

(D10) The only way of testing the applicability of any theory is to build a software system based on the theory, to deploy a set of applications based on the theory, and to report on experiments based on those applications. Thus, an implementation *must be validated* by a set of deployed applications.

1.6 A Birdseye View of This Book

This book is organized as follows.

Chapter 2 introduces the reader to the overall architecture of the proposed *IMPACT* framework. It explains the issues involved in designing the architecture, what alternative

architectures could have been used, and why certain design choices were made. It explains the architecture of individual agents, as well as the architecture of the agent infrastructure, and how the two “fit” together, using the STORE, CFIT, and CHAIN examples to illustrate the concepts.

Chapter 3 explains the *IMPACT* Service Description language using which an agent may specify the set of services that it offers. We describe the syntax of the language and specify the services offered by various agents in the STORE, CFIT, and CHAIN examples using this syntax. We further specify how requests for *similar* services are handled within this framework. We explain existing alternative approaches, and describe the advantages of disadvantages of these approaches when compared to ours.

Chapter 4 shows how agents may be built on top of legacy data, using a basic mechanism called a *code call condition*. We will show how such access methods may be efficiently implemented and we will describe our implementation efforts to date to do so. As in other cases, the STORE, CFIT, and CHAIN examples will be revisited here.

Chapter 5 describes the implementation of *IMPACT* Servers. These are programs that provide the “infrastructural” services needed for multiple agents to interact, including yellow pages and other services described in detail in Chapter 3. This chapter explains how to access these servers, how they were implemented, and how agents may interact to them. A theorist may wish to skip this chapter, but an individual seeking to implement an agent system may find this chapter very useful.

Chapter 6 builds on top of Chapter 4 and shows how an agent’s action policies may be declaratively specified. Such declarative policies must encode what the agent is permitted to do, what it is forbidden from doing, what it is obliged to do, and what in fact, it does, given that the agent’s data structures reflect a “current state” of the world. We show that the problem of determining how to “act” (which an agent must make continuously) in a given agent state may be viewed as computing certain kinds of objects called “status sets.” In this chapter, we assume a frozen instant of time, and make very few assumptions about “states.” These concepts are illustrated though the STORE, CFIT, and CHAIN examples.

In Chapter 7 we argue that an agent’s state may (but does not have to!) contain some information about the agent’s beliefs about other agents. This is particularly useful in adversarial situations where agent *a* might want to reason about what agent *b*’s state before deciding what to do. The theory of Chapter 4 is extended to handle such meta-reasoning. There is also another example introduced, the RAMP example, which was particularly designed agents reasoning about beliefs.

In Chapter 8, we extend the theory of Chapter 6 in yet another direction—previously, a “frozen” instant of time was assumed. Of course, this is not valid—we all make decisions today on what we will do tomorrow, or day after, or next month. We create schedules for

ourselves, and agents are no different. This chapter describes an extension of the theory of Chapter 7 to handle such temporal reasoning.

In Chapter 9, we add a further twist, increasing the complexity of both Chapter 7 and 8, by assuming that an agent may be *uncertain* both about its beliefs (about the state of the world, as well as its beliefs about other agents). The theory developed in previous chapters is extended to handle this case.

In Chapter 10, we revert to our definition of states and actions, and examine specific data structures that an agent must maintain in order to preserve security, and specific actions it can take (relative to such data structures) that allow it to preserve security. We further explore the relationship between actions taken by individual agents and the data structures/algorithms built into the common agent infrastructure, with a view to maintaining security.

In Chapter 11, we develop a body of complexity results, describing the overall complexity of the different languages developed in preceding chapters. The chapter starts out with a succinct summary and interpretation of the results—a reader interested in the “bottom line” may skip the rest of this chapter.

In Chapter 12, we identify efficiently computable fragments of agent programs, and provide polynomial algorithms to compute them. We explain what can, and what cannot, be expressed in these fragments. We then describe *IADe*—the *IMPACT* Agent Development Environment, that interested users can use to directly build agents in *IMPACT*, as well as build multi-agent systems in *IMPACT*. We will report on experiments we have conducted with *IMPACT*, and analyze the performance results we obtain.

In Chapter 13, we will describe in detail, an integrated logistics application we have built within the *IMPACT* framework for the US Army.

Finally, in Chapter 14, we will revisit the basic goals of this book—as described in Chapters 1 and 2, and explain how we have accomplished them. We identify the strengths of our work, as well as shortcomings that pave the way for future research by us, and by other researchers.

1.7 Selected Commercial Systems

There has been an increase in the number of agent applications and agent infrastructures available on the Internet. In this section, we briefly mention some of these commercial systems.

Agents Technologies Corp.’s *Copernic 98* (<http://www.copernic.com/>) integrates information from more than 110 information sources.

Dartmouth College’s *D’Agents* project (<http://www.cs.dartmouth.edu/~agent/>) supports applications that require the retrieval, organization, and presentation of distributed information in arbitrary networks.

Firefly's *Catalog Navigator* (<http://www.firefly.net/company/keyproducts.fly>) allows users to add preference and general interest-level information to each customer's personal profile, hence providing more personalized service.

General Magic's *Odyssey* (<http://www.genmagic.com/agents/>) provides class libraries which enable people to easily develop their own mobile agent applications in Java. It also includes third party libraries for accessing remote *CORBA* objects or for manipulating relational databases via *JDBC*.

IBM's *Aglets* provide a framework for development and management of mobile agents. (<http://www.tr1.ibm.co.jp/aglets/>). An aglet is a Java object having mobility and persistence and its own thread of execution. Aglets can move from one Internet host to another in the middle of execution, (Lande and Osjima 1998). Whenever an aglet moves, it takes along its program code and data. Aglets are hosted by an Aglet server, as Java applets are hosted by a Web browser.

Microelectronics and Computer Technology Corporation's *Distributed Communicating Agents* (DCA) for the Carnot Project (<http://www.mcc.com/projects/carnot/DCA.html>) enables the development and use of distributed, knowledge-based, communicating agents. Here, agents are expert systems that communicate and cooperate with human agents and with each other.

Mitsubishi Electric ITA Horizon Systems Laboratory's *Concordia* (<http://www.meitca.com/HSL/Projects/Concordia/>) is a full-fledged framework for development and management of network-efficient mobile agent applications for accessing information anytime, anywhere, and on any device supporting Java. A key asset is that it helps abstract away the specific computing or communication devices being used to access this data.

ObjectSpace's *Voyager* (<http://www.objectspace.com/voyager/>) allows Java programmers to easily construct remote objects, send them messages, and move objects between programs. It combines the power of mobile autonomous agents and remote method invocation with *CORBA* support and distributed services.

Oracle's *Mobile Agents* (http://www.oracle.com/products/networking/mobile_agents/html/index.html) is networking middleware designed to facilitate connectivity over low bandwidth, high latency, occasionally unreliable, connections. It may be used to help provide seamless data synchronization between mobile and corporate databases.

Softbot (software robot) programs (<http://www.cs.washington.edu/research/projects/softbots/www/projects.html>) are intelligent agents that use software tools and services on a person's behalf. They allow a user to communicate what they want accomplished and then dynamically determine how and where to satisfy these requests.

Stanford's *Agent Programs* (<http://www-ksl.stanford.edu/knowledge-sharing/agents.html>) provide several useful agent-related utilities such as a content-based router

(for agent messages), a matchmaker (w.r.t. agent interests), and many more. They follow the KIF/KQML protocols (Neches, Fikes, Finin, Gruber, Patil, Senator, and Swarton 1991; Genesereth and Fikes 1992; Labrou and Finin 1997a; Finin, Fritzon, McKay, and McEntire 1994; Mayfield, Labrou, and Finin 1996; Finin et al. 1993) for knowledge sharing.

UMBC's *Agent Projects* (<http://www.cs.umbc.edu/agents/projects/>) include several applications such as Magenta (for the development of agent-based telecommunication applications), AARIA (for autonomous agent based factory scheduler at the Rock Island Arsenal), etc. UMBC also maintains descriptions of several projects using KQML (<http://www.csee.umbc.edu/kqml/software/>), a Knowledge Query and Manipulation Language for information exchange.

Some other sites of interest include the *Agent Society* (<http://www.agent.org/>) and a site http://csvax.cs.caltech.edu/~kiniry/projects/papers/IEEE_Agent/agent_paper/agent_paper.html, which surveys *Java Mobile Agent Technologies*.