9 Industrial and Practical Applications of DAI

H. Van Dyke Parunak

9.1 Introduction

Successful application of agents (as of any technology) must reconcile two perspectives. The researcher (exemplified in the preceding chapters) focuses on a particular capability (e.g., communication, planning, learning), and seeks practical problems to demonstrate the usefulness of this capability (and justify further funding). The industrial practitioner has a practical problem to solve, and cares much more about the speed and cost-effectiveness of the solution than about its elegance or sophistication. This chapter attempts to bridge these perspectives. To the agent researcher, it offers an overview of the kinds of problems that industrialists face, and some examples of agent technologies that have made their way into practical application. To the industrialist, it explains why agents are not just the latest technical fad, but a natural match to the characteristics of a broad class of real problems.

This chapter is both broader and narrower than its title suggests. It is broader because it includes selected development projects that are not yet industrial strength, but embody industrially important concepts or are being conducted in a way likely to lead to deployable technology. It is narrower in that it emphasizes agent applications in manufacturing and physical control over other fielded industrial applications such as Web-based information-gathering agents ([50]; Appendix G of [12]), network management, or business planning agents (e.g., the ADEPT project [45, 47, 48, 60]). This focus has three motivations: I am better acquainted with the domain of manufacturing and control, the problems of interfacing agents to the environment are more challenging, and the evidence of success or failure is clearer when a system must directly confront the laws of physics.

Section 9.2 describes the main industrial motivations for choosing an agent architecture for a particular problem. Section 9.3 describes the concept of a system life cycle. Section 9.4 and uses this concept to organize case studies of industrial agent-based systems, and Section 9.5 uses it to examine the process of constructing an industrial system. Section 9.6 reviews some development tools that will hasten deployment of agent technology in industry. Section 9.7 summarizes some basic insights.

9.2 Why Use DAI in Industry?

Agents are not a panacea for industrial software. Like any other technology, they are best used for problems whose characteristics require their particular capabilities. Agents are appropriate for applications that are modular, decentralized, changeable, ill-structured, and complex [44]. In some cases, a problem may naturally exhibit or lack these characteristics, but many industrial problems can be formulated in different ways. In these cases, attention to these characteristics during problem formulation and analysis can yield a solution that is more robust and adaptable than one supported by other technologies.

Modular. As defined in more detail in Chapter 1, agents are pro-active objects, and share the benefits of modularity enjoyed by object technology. They are best suited to applications that fall into natural modules. An agent has its own set of state variables, distinct from those of the environment. Some subset of the agent's state variables is coupled to some subset of the environment's state variables to provide input and output. An industrial entity is a good candidate for agent-hood if it has a well-defined set of state variables that are distinct from those of its environment, and if its interfaces with that environment can be clearly identified.

The state-based view of the distinction between an agent and its environment suggests that functional decompositions are less well suited to agent-based systems than are physical decompositions. Functional decompositions tend to share many state variables across different functions. Separate agents must share many state variables, leading to problems of consistency and unintended interaction. A physical decomposition naturally defines distinct sets of state variables that can be managed efficiently by individual agents with limited interactions. The choice between functional and physical decomposition is often up to the system analyst. Emphasizing the physical dimension enables more modular software. Because the agent characterizes a physical entity, that entity can be redeployed with minimal changes to the agent's code. As a result, the cost of software reconfiguration drops dramatically, and reusability increases.

Decentralized. An agent is more than an object; it is a pro-active object, a bounded process. It does not need to be invoked externally, but autonomously monitors its own environment and takes action as it deems appropriate. This characteristic of agents makes them particularly suited for applications that can be decomposed into stand-alone processes, each capable of doing useful things without continuous direction by some other process.

Many industrial processes can be organized in either a centralized or a decentralized fashion. Centralized organizations go back to the imperial governments of ancient Egypt, Assyria, China, and Babylon, with their focus on a central demigod and an elaborate bureaucracy to manage the flow of control down and information back up. The popularity of this structure can be traced through the army of

Alexander the Great, the Roman legions, and the rival empires of pre-modern Europe down to the structure of modern Fortune 500 companies and industrial control architectures [2].

There is an alternative approach. The power of decentralization has been made clear in recent years in the contrast in performance between a centralized economic system (the former Soviet Union) and a decentralized one (free-market capitalism). In fact, a European observer suggests that one of the forces leading to the growing popularity of multiagent systems is "the rise of the American style of liberalism and individualism" [80].

Modern industrial strategists seek to eliminate excessive layers of management and push decision-making down to the very lowest level, and are developing the vision of the "virtual enterprise," formed for a particular market opportunity from a collection of independent firms with well-defined core competencies [58]. It is increasingly common for the manufacturer of a complex product to purchase half or even more of the content in the product from other companies. For example, an automotive manufacturer might buy seats from one company, brake systems from another, air conditioning from a third, and electrical systems from a fourth, and manufacture only the chassis, body, and powertrain in its own facilities. The suppliers of major subsystems (such as seats) in turn purchase much of their content from still other companies. As a result, the "production line" that turns raw materials into a vehicle is a network, or "supply chain," of many different firms. Agent-based architectures are an ideal fit to such an organizational strategy.

Changeable. Agents are well suited to modular problems because they are objects. They are well suited to decentralized problems because they are pro-active objects. These two characteristics combine to make them especially valuable when a problem is likely to change frequently. Modularity permits the system to be modified one piece at a time. Decentralization minimizes the impact that changing one module has on the behavior of other modules.

Modularization alone is not sufficient to permit frequent changes. As Figure 9.1 suggests, in a system with a single thread of control, changes to a single module can cause later modules, those it invokes, to malfunction. Decentralization decouples the individual modules from one another, so that errors in one module impact only those modules that interact with it, leaving the rest of the system unaffected. (The original version of this figure was created by Seiichi Yaskawa of Yaskawa Electric Corporation, Tokyo, Japan, and is used with his kind permission.)

From an industrial perspective, the ability to change a system quickly, frequently, and without damaging side effects is increasingly important to competitiveness. In manufacturing, the product that pleases the most customers has a tremendous advantage. One of the most effective means to determine the features that customers like is to turn out as many different product variations as quickly as possible, sampling customer response and adjusting new offerings accordingly. This strategy is responsible for the precipitous drop in the time-to-market for many products. The time from product concept to first production in automotive used to be 60

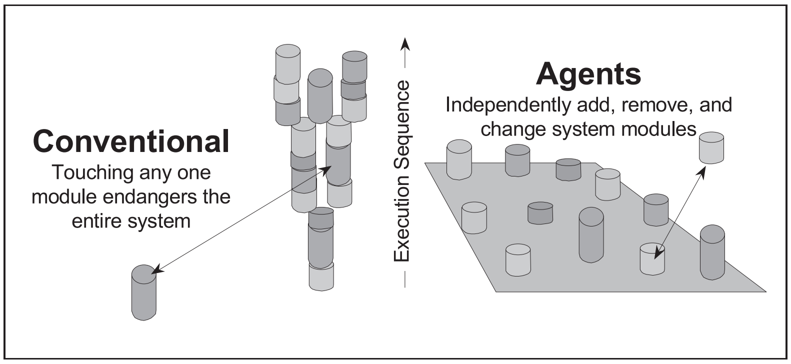

Figure 9.1 Modularity + Decentralization → Changeability

months. Now world-class performance requires 30 months, and some vehicles have been produced in even less time. Much of the cost of a new manufacturing facility is in its software. Agent-based architectures permit reuse of much existing code and self-configuration of large portions of the system, reducing both the cost and the time needed to bring up a new factory.

Ill-structured. An early deliverable in traditional systems design is an architecture of the application, showing which entities interact with which other entities and specifying the interfaces among them. For example, installation of a conventional system for electronic data interchange (EDI) among trading partners requires that one know the providers and consumers of the various goods and services being traded, so that orders can be sent to the appropriate parties. Sometimes, determining this information in advance is extremely difficult or even impossible. Consider an electronic system to support open trading, where orders are made available to any qualified bidder. Requiring the system designer to specify the sender and recipient of each transaction would quickly lead to “paralysis by analysis.” From a traditional point of view, this application is ill structured. That is, not all of the necessary structural information is available when the system is designed.

Agents naturally support such an application. The fundamental distinction in an agent’s view of the world is between “self” and “environment.” “Self” is known and predictable, while “environment” can change on its own within limits. Other agents are part of this dynamic, changing environment. Depending on the complexity of individual agents, they may or may not model one another explicitly. Instead of specifying the individual entities to be interconnected and their interfaces with one another, an agent-based design need identify only the classes of entities in the system and their impact on the environment. Because each agent is designed to interact with the environment rather than with specific other agents, it can interact appropriately with any other agent that modifies the environment within the range of variation with which other agents are prepared to deal.

Some applications are intrinsically under-specified and thus ill structured, and agents offer the only realistic approach to managing them. Even where more detailed structural information is available, the wiser course may be to pretend that it isn't. A system that is designed to a specific domain structure will require modification if that structure changes. Agent technology permits the analyst to design a system to the classes that generate a given domain structure rather than to that structure itself, thus extending the useful life of the resulting system and reducing the cost of maintenance and reconfiguration.

Complex. One measure of the complexity of a system is the number of different behaviors it must exhibit. For example, a manufacturing job shop might produce a given part in several different ways, depending on which machines are used and in which order. The number of possible behaviors in this simple example depends exponentially on the number of different machines in the shop. For a shop with only a few machines, one might code a separate subroutine for each possible routing, but this approach quickly becomes prohibitive as the shop grows.

This example shows combinatorial complexity. The number of different interactions among a set of elements increases much faster than does the number of elements in the set. By mapping individual agents to the interacting elements, agent architectures can replace explicit coding of this large set of interactions with generation of them at run-time. Consider 100 agents, each with ten behaviors, each behavior requiring 20 lines of code. The total amount of software that has to be produced to instantiate this system is 20,000 lines of code, an extremely modest undertaking. But the total number of behaviors in the repertoire of the resulting system is on the order of ten for the first agent, times ten for the second, times ten for the third, and so forth, or 10^{100}, an overwhelmingly large number. Naturally, not all of these will be useful behaviors, and one can imagine pathological agent designs in which none of the generated behaviors will be appropriate. However, appropriately designed agent architectures can move the generation of combinatorial behavior spaces from design-time to run-time, drastically reducing the amount of software that must be generated and thus the cost of the system to be constructed.

Modification of a system during its life can increase its complexity as well as making it ill structured. By adopting an agent approach at the outset, systems engineers can provide a much more robust and adaptable solution that will grow naturally to meet business needs.

9.3 Overview of the Industrial Life-Cycle

Industrial people tend to view what they do in terms of a life cycle, made up of a series of stages: requirements analysis, design, implementation and deployment, operation, logistics and maintenance, and decommissioning. Any industrial activity follows such a pattern, whether it be building a product, putting in place the process for making a product, supplying a service, or creating a piece of infrastructure.

The life cycle perspective raises two questions about industrial multiagent systems. First, to what stages in the life cycle of an industrial activity (say, an automobile) have agents been applied? Second, since an industrial agent-based system will itself be constructed according to a life cycle, what constraints does the industrial environment place on each of the life-cycle phases of such a system?

In this exposition, the term "project" represents a specific system or activity. Two projects are discussed: a physical system (a new automobile), and a software system (a new factory scheduling system). Figure 9.2 shows how the physical system bifurcates at the design phase into two systems, one concerned with the product itself, the other concerned with the process that manufactures the product. A generic life cycle has eight phases, some of which may not be appropriate in a given project.

Requirements Definition defines the set of needs or requirements that the project must satisfy. The focus is on why an effort is needed in the first place, not on *what* the project will do or *how* it will do it.

- Physical: Market analysis reveals that we are losing sales to competitors who are offering sport utility vehicles, a niche in which we currently have no product offering.
- Software: We have benchmarked our production facilities against world class performance, and found that we are below the 75% level in every major category, including throughput, machine utilization, order tardiness, and work-in-process levels.

Positioning defines the project's relationship to other projects in the enterprise. This phase identifies potential overlaps, synergies, or conflicts among different projects early enough that their impact can be managed.

- Physical: Our current product divisions are luxury auto, economy auto, minivan, and light truck. The minivan and light truck divisions seem the best candidates to host the new offering. Further study shows that we are aiming for a consumer market, not an industrial one, so the new vehicle is positioned as a new product in the minivan division.
- Software: Shop-floor control can be approached from the perspective either of controls (a bottom-up view) or of manufacturing information systems (a top-down view). In our company, the information department is notoriously insensitive to plant needs, and their data is usually wrong, so we have no confidence that a scheduling project that grows out of our existing information systems will solve the requirements. However, the controls group has been remarkably successful in solving a wide range of integration problems, so we will try to address our problems from the controls perspective.

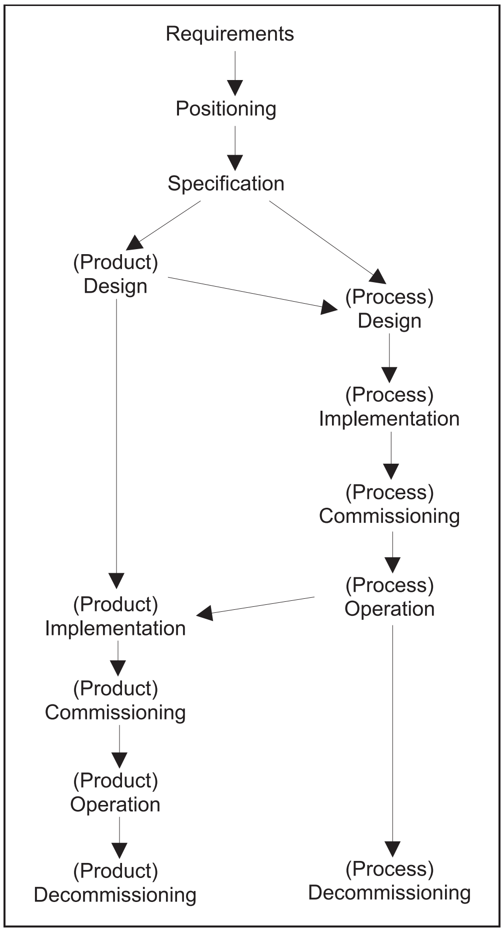

Figure 9.2 The life cycle for physical products bifurcates.

Specification spells out the functions that the project will support. The specification tells what the project will do, but not how it does it. The functions in a successful specification will satisfy the needs identified in Requirements Definition and interface appropriately with other relevant components of the enterprise identified in Positioning.

- Physical: We benchmark the performance of our competitors' offerings to determine what customers do and do not like, and to identify features we can add to differentiate our product in the marketplace. The result is a list of the features and performance characteristics of the new vehicle.
- Software: A collection of shop-floor war stories highlights a set of issues that can explain the poor performance, including no way to schedule preventive

maintenance (leading in turn to reduced maintenance and increased machine failure), operating policies that permit upstream workstations to produce parts for which there is no downstream demand, release of jobs to the floor before both raw materials and tooling are available, and job classifications that prevent operators from helping one another as demands shift across the factory.

Design maps the functions ("what") identified in Specification to implementation strategies that tell how the project will be executed to provide those functions.

- Physical: The product engineering department develops a design for the new vehicle, including chassis, seating system, powertrain, suspension, climate control, sound system, and beverage cooler. Concurrently, the process engineering department designs the factory that will manufacture the new vehicle. At this point the automotive project actually becomes two projects: one to produce the vehicle, the other to produce the process that will make the vehicle.
- Software: The manufacturing systems group decides to adopt an agent-based approach to shop scheduling. It identifies the classes of agents that will be required, and refines these classes and their interactions through role-playing and simulation.

Implementation is the phase of the life cycle in which the system is actually constructed. If the project is an activity rather than a system, it may move directly form Design to Commissioning without an intervening Implementation.

- Physical (Process): Purchasing negotiates contracts for the equipment needed to construct the new vehicle. A plant is selected to house the new line, the old equipment is removed, and the new equipment is installed.
- Physical (Product): Purchasing negotiates contracts for the raw materials and preassembled subsystems that will be purchased from vendors
- Software: The systems group codes the agents that will make up the new scheduling system. The modular decentralized nature of agent-based software makes it possible to extend some design activities, such as system simulation, into implementation by running newly coded real agents against a simulation of the part of the system not yet implemented. The emergent nature of agent-based systems makes this approach necessary to avoid unexpected global behaviors.

In **Commissioning**, the project is placed into use. Commissioning usually includes system shakedown, training activities, and transition of operations from previous systems or methods.

- Physical (Process): The factory produces its first sports utility vehicle.
- Physical (Product): Each unit of the product is commissioned when a dealer sells it to a customer.
- Software: The scheduling agents are released onto the shop floor.

Operation maintains the project in regular productive use. It is during this phase that the project actually satisfies the needs identified during Requirements Definition. Operation includes three specific activities: routine operation, maintenance and repair, and incremental upgrading. In the life cycle of a product, this phase also includes customer support and maintenance.

- Physical (Process): The factory continues to produce vehicles.
- Physical (Product): The dealer network services the vehicles already in the field.
- Software: The systems group adjusts the behavior of individual agents based on feedback from the operators and changes in the firm's business environment.

Decommissioning removes the project from service, either because the needs it satisfied no longer exist or because a replacement project is about to be commissioned. The growing importance of ecologically friendly or "green" manufacturing is placing increasing emphasis on this phase as the point at which reuse or recycling is applied.

- Physical (Process): After about ten years, this model is phased out, and the factory and equipment that produced it are reconfigured for a new product.
- Physical (Product): Vehicles that have completed their useful life are recycled.
- Software: Because the scheduling system is agent-based, there is no sharp line when it is decommissioned as a system. Individual agents are replaced over the years as equipment changes and new functionality is required, but the changes are incremental.

9.4 Where in the Life Cycle Are Agents Used?

In principle, agents can support many different stages in the life cycle of a system or product. For example, agents might help design a new vehicle, operate the plant that manufactures it, and maintain it when it fails. This section begins by outlining a series of questions that are useful in comparing different agent applications. Then it describes three areas in which agents have been used effectively: product design, process operation at the planning and scheduling level, and process operation at the lower level of real-time equipment control.

9.4.1 Questions that Matter

When comparing different agent-based applications, some questions arise repeatedly. This section groups these questions in three categories: those that pertain to individual agents, those that concern the community of agents, and a single question dealing with the maturity of the application.

The building blocks of agent-based systems are the individual agents. Three questions are important here. Chapter 1 explores a number of these details in greater depth.

What in a system becomes an agent? Classical software engineering techniques lead many systems designers toward "functional decomposition." For example, manufacturing information systems typically contain modules dedicated to functions such as "scheduling," "material management," and "maintenance," suggesting that these functions should be assigned to distinct agents [6, 32]. The functional approach is well suited to centralized systems, but unprecedented in naturally occurring distributed systems, which divide agents on the basis of distinct entities in the physical world rather than functional abstractions in the mind of the designer [71]. Experience with agent-based prototypes supports this principle, with two exceptions. Most industrial agent applications are additions to existing systems, and functionally oriented *legacy systems* may be most easily attached by encapsulating them as (functional) agents. A *watchdog agent* may usefully monitor the behavior of a population of physical agents for important system states that local agents cannot perceive.

How Does each agent model the world? Any agent that functions in a changing world must model that world internally [40]. However, agents differ in the sophistication of the knowledge representation and reasoning they use for this task. Some agents model aspects of their world explicitly, so that they can reason about the model. In other agents, these models are hard-wired and often distributed throughout the agent's architecture [26]. In addition to the implicit-explicit distinction, agents differ in the scope of what they model (for instance, whether they individuate other agents or not) and whether they model the world as it is now, or as the agent wishes the world to be. The BDI architecture discussed in Chapter 1 recommends explicit models that include both the present (beliefs) and the future (desires).

How are agents structured internally? The different agents in a system may be identical, heterogeneous, or sharing some common modules and differing in others. They may or may not remember past states, and their internal code may or may not change over time.

The next five questions have to do with the community that the agents form. Chapter 2 discusses many of these issues in more detail.

How many agents are there? Both the number of different agent species and the total number of individual agents are important characteristics of a given system, as well as whether the agent population can change while the system is running.

What communication channels do agents use? The channels through which information moves from one agent to another can differ in medium (the shared physical environment vs. a digital network), addressing (broadcast, subject-based, agent-to-agent), whether messages persist after being sent, and locality (whether agents need to move "close" to one another in order to exchange messages).

What communications protocols do agents use? A communications protocol determines how conversations among agents are structured. Some agents simply give orders to one another and expect them to be received. Others vote, negotiate, or engage in more complex dialogues based on speech-act theory.

How is the configuration of agents relative to one another established? The configuration of an agent community describes the immediate acquaintances of each agent and the resulting topology over which information and material move among them. This topology may be set in advance by the system implementer and remain unchanged as the system operates, or the agents may be able to discover new relationships and reconfigure themselves while running.

How do agents coordinate their actions? Agents are autonomous in that they do not have to be invoked in order to execute. However, in a useful system they are not anarchical, but coordinate with one another. In hierarchical coordination, commands flow down from higher levels and status information flows back up. In egalitarian or heterarchical coordination [37], coordination emerges from the dynamics of agent interaction, through mechanisms such as dissipative fields (currency-based markets [15], pheromones in insect societies) or constraint propagation.

The final question is **"How mature is the application?"** It is useful to distinguish six levels of maturity:

1. Modeled: The system exists as an architecture or a theoretical model.
2. Emulated: The system has been demonstrated against a simulation of its intended domain environment.
3. Prototype: The system has been demonstrated on real domain hardware, but in a controlled laboratory environment.
4. Pilot: The system has been demonstrated in a commercial environment.
5. Production: The system is used in regular commercial practice.
6. Product: The system is sold and supported as a commercial product.

9.4.2 Agents in Product Design

Design systems help teams of designers, often in different locations and working for different companies, to design the components and subsystems of a complex product, using many different analysis tools. As suppliers take increasing responsibility for the detailed design of the subsystems they supply, design becomes increasingly decentralized.

Designers begin with a picture of what is required but no details on how it is to be produced. Often the "what" that is desired turns out to be prohibitively

System	What is an Agent?	Noteworthy Technology
ACDS (Automated Configuration Design Service) [19]	Catalogs of pre-defined parts; constraints among components; the overall system; a bid monitor.	Distributed constraint management
PACT (Palo Alto Collaborative Testbed) [18]	Pre-existing design tools; facilitators to translate between tools and a common language	A common language between agents
RAPPID (Responsible Agents for Product-Process Integrated Design) [72, 76]	Designers (representing components and subsystems of the product); design variables	Set-based reasoning and market protocols among designers

Table 9.1 Design systems.

expensive when the "how" is understood in more detail, leading to frequent changes in the design. The more ambitious the product vision, the less well its structure is understood at the outset, and the more valuable reconfigurable agents are to represent the various components, designers, and tools. The increased complexity embodied in modern products also favors the combinatorial benefits of an agent-based system. State-of-the-art agent concepts have been demonstrated in three design systems at the prototype level of maturity. Each of these systems decomposes the world into agents in a different way, as summarized in Table 9.1.

RAPPID illustrates how agents can help human designers coordinate their work more effectively. Figure 9.3 illustrates how different people are responsible for the components and subsystems that make up a product. Conflicts arise when different teams disagree on the relation between the characteristics of their own functional pieces and the characteristics of the entire product. Some conflicts are within the design team: How much of a mechanism's total power budget should be available to the sensor circuitry, and how much to the actuator? Other conflicts set design against other manufacturing functions: How should one balance the functional desirability of an unusual machined shape against the increased manufacturing expense of creating that shape?

It is easy to represent how much a mechanism weighs or how much power it consumes, but there is seldom a disciplined way to trade off (say) weight and power consumption against one another. The more characteristics are involved in a design compromise, the more difficult the trade-off becomes. The problem is the classic dilemma of multivariate optimization. Analytical solutions are available only in specialized and limited niches. In current practice such trade-offs are sometimes supported by processes such as Quality Functional Deployment [38] or resolved politically, rather than in a way that optimizes the overall design and its manufacturability. The problem is compounded when design teams are distributed across different companies.

RAPPID explores two innovative techniques for coordinating the actions of

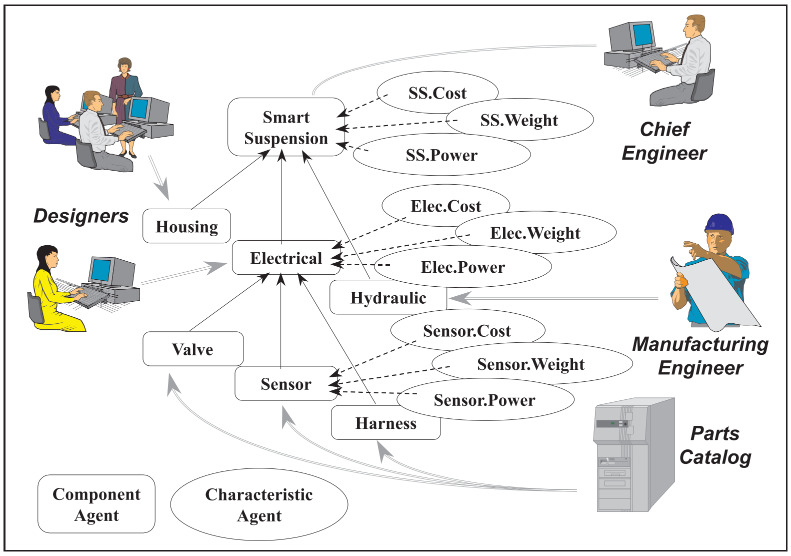

Figure 9.3 The RAPPID ecosystem includes both components and characteristics.

different agents (designers): *market dynamics* and *set-based reasoning* [77, 78].

In RAPPID, designers buy and sell the various characteristics of a design. Each characteristic agent is a computerized agent that maintains a marketplace in that characteristic. In the current implementation, the agents representing components are interfaces for human designers, who bid in these markets to buy and sell units of the characteristics. A component that needs more latitude in a given characteristic (say, more weight) can purchase increments of that characteristic from another component, but may need to sell another characteristic to raise resources for this purchase. In some cases, analytical models of the dependencies between characteristics help designers estimate their relative costs, but even where such models are clumsy or nonexistent, prices set in the marketplace define the coupling among characteristics.

To drive the design process toward convergence, RAPPID uses set-based reasoning. Most design in industry today follows a point-based approach, in which the participating designers repeatedly propose specific solutions to their component or subsystem. The chief engineer is expected to envision the final product at the outset, specifying to the designers what volume in design space it should occupy and challenging them to fit something into that space. Inevitably, as illustrated in Figure 9.4, some of the chief engineer's assumptions turn out to be wrong, requiring designers to reconsider previous decisions and compromise the original vision. This approach is analogous to constraint optimization by backtracking. Because mechanisms for disciplined backtracking are not well developed in design methodology, this approach usually terminates through fatigue or the arrival of a critical market deadline, rather than through convergence to an optimal solution.

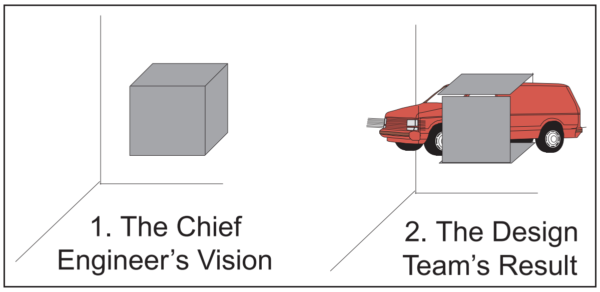

Figure 9.4 Point-based design requires backtracking.

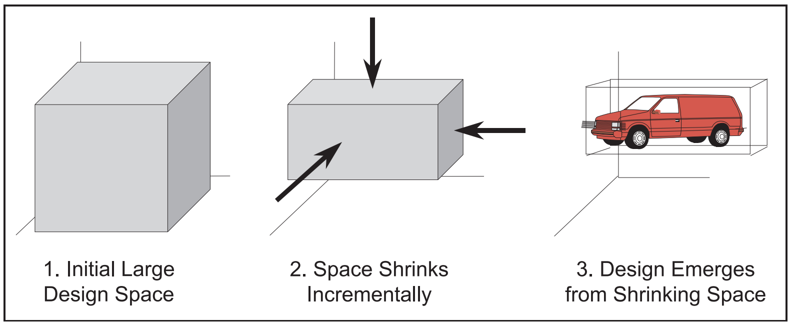

Figure 9.5 Set-based design converges to the solution.

Toyota has pioneered another approach, set-based design [89]. In this approach, illustrated in Figure 9.5, the chief engineer's task is not to guess the product's location in design space, but to guide the design team in a process of progressively shrinking the design space until it collapses around the product. Each designer shrinks the space of options for one component in concert with the other members of the team, all the while communicating about their common dependencies. This approach directly reflects consistency algorithms for solving constraint problems. If the communications among team members are managed appropriately, the shrinking design space drives the team to convergence.

Here is how RAPPID answers the questions introduced earlier in this section.

Agent Mapping and Modeling: Component agents are computer-assisted humans and thus maintain extensive as-is and to-be models of the other agents and the non-agented environment. Characteristic agents model the component agents that have an interest in them, and use these models to recommend future action.

Agent Structure: Characteristic agents are structurally identical to one another. Their code does not change over time, but they do aggregate information from recent bids as a guide to future activity. Component agents vary as widely as the humans they represent. If a component is to be selected from a catalog, RAPPID

	Planning & Scheduling	Control
Time Constants	Human-scale (minutes and longer)	Electronic-scale (seconds and shorter)
Kind of Information	Includes symbols; involves semantics	Mostly numerical

Table 9.2 Comparing planning and control.

provides a catalog module that automates much of the problem of selecting the appropriate offering, and thus subsumes some of the functionality of ACDS. If a component is to be designed from the ground up, the RAPPID interface hides the idiosyncrasies of the various tools the human may use in the design process, and thus subsumes some of the functionality of PACT.

Population: A realistic application of RAPPID will have one or two dozen component agents and on the order of a hundred characteristic agents. In the current implementation, agents are not created, destroyed, divided, or fused during operation, but as the system matures, designers will need a way to add both component and characteristics agents to the community as a design is refined.

Communication Channels and Protocols: Agents communicate digitally, and currently use point addressing. Messages do not persist outside of agents, and agents do not move over the network. RAPPID uses a fixed market protocol, but also provides for the humans behind component agents to communicate directly with one another using Standard Legacy-Oriented Work Habits (SLOWH mechanisms).

Configuration: The initial configuration of component agents and characteristic agents is defined when the system is initialized, but component agents can engage in markets for other characteristics as the system runs.

Coordination: RAPPID combines dissipative and constraint-based egalitarian coordination.

Maturity: RAPPID has been piloted in the high-level design of a military vehicle at the U.S. Army's Tank and Automotive Command (TACOM) at Warren, MI.

9.4.3 Agents in Planning and Scheduling

Both this section and the next deal with the problem of monitoring a system's trajectory through state space over time and adjusting operating parameters to make that trajectory satisfy some overall criterion. The difference between the two is one of time constants and the kind of information manipulated, as summarized in Table 9.2. Planning and scheduling is longer-term, usually on a scale that humans can handle, and involves the manipulation of concepts through semantically-grounded symbols. Control handles the detailed real-time interface with the world, and usually happens too fast for direct human supervision.

The fundamental challenge in applying agents to both planning and control is satisfying a global criterion on the basis of parallel local decisions. In spite of the natural benefit that centralization has in dealing with control criteria, the cases in these two sections show that many users have found agents an even better approach. Operational systems must be maintained, and it is much easier and safer to maintain a set of well-bounded modules than to make changes to a large monolithic program. The move toward supply chains means that the manufacturing system is geographically distributed, and agent decentralization reduces communication bottlenecks and permits local parts of the enterprise to continue operation during temporary lapses in connectivity. Competitiveness increasingly depends on adjusting a system's operation frequently to track customer requirements, benefiting from the changeability of agent systems. The ability of agents to deal with ill-structured systems is less important in the operation of an engineered system than in its design. However, the ability of agents to deal with dynamically changing structures means that computers can now be applied to manage systems (such as networks of trading partners) that formerly required extensive manual attention. The increased complexity that agents can manage also extends the scope of operational problems to which they can be applied.

Table 9.3 summarizes four examples of planning and scheduling systems. The Daewoo scheduling system produced by Metra Corp. is a mature example (in regular production use) of the most promising approach to agent-based scheduling and control. It schedules the press shop at Daewoo Motors' integrated automobile production facility in Korea.

Most of the exterior components of an automobile, and many structural components as well, are manufactured by stamping sheet metal between shaped metal dies in hydraulic presses exerting hundreds or thousands of tons of pressure. The process of setting up and running such a press is daunting. Sheet metal arrives in coils that must be unrolled and cut into blanks before being stamped. Different parts require different kinds of sheet metal, from different coils. The dies weigh thousands of pounds, and must be transported from a storage area and aligned precisely with the press before parts can be made. Dies wear with use, requiring periodic refurbishing. The type of part being produced (and thus the setup) must change frequently to provide the vehicle assembly operation with the right components for the vehicles currently being manufactured.

The Daewoo shop supplies all stamped body parts for five different car models, as well as parts for several off-site assembly plants. The shop operates three shifts per day and produces more than 500 different parts, using more than 2000 dies. Efficient operation requires careful scheduling of the presses, sheet metal stock, and dies.

Agent Mapping: Research on agent-based factory planning and scheduling differs widely on what is represented as an agent. ISCM assigns an agent to each traditional manufacturing function (such as Order Acquisition, Logistics, Scheduling, Resource Management, Dispatching, Transportation, and Plant Management). Evidence from

System	What is an Agent?	Noteworthy Technology
AARIA (Autonomous Agents for Rock Island Arsenal) [4, 73]	Resources (e.g., machines, tooling, and operators), part types, unit processes, management, parts, and engagements between a unit process and its resources	Market-driven inter-agent coordination; density-based least-commitment scheduling of resources
Daewoo [14]	Task agents (work orders), resource agents (machines), service agents (per community: bidding, constraint propagation, meta-agent)	Market-driven coordination; hierarchical aggregation of agents into communities
ISCM (Integrated Supply Chain Management) [31]	Functional agents: Logistics, Order Acquisition, and Transportation agents at enterprise level; Plant Manager, Resource Management, Dispatching, and Scheduler agents at plant level	Traditional AI within individual functional agents
LMS (Logistics Management System) [29, 30]	A critic for each production characteristic (Serviceability, Daily Planned Output, Downstream Pull, and Tool Charge, Characteristics, & Utilization); and a judge to combine votes	The judge combines votes from critics based on their individual objectives.

Table 9.3 Planning and scheduling systems.

natural systems suggests that it may be more effective to assign agents to physical entities in the system. Even here there is considerable variation. Agents have represented levels in a hierarchical decomposition of the factory [11, 67, 87], resources [3, 39, 74, 84], and parts [20, 53]. AARIA has developed a comprehensive ontology, summarized in Figure 9.6. It includes both agents that persist from one operation to another (such as part types, unit processes, and resources) and agents with much shorter lifetime (including individual parts and engagements between a unit process and the resources it requires). In terms of this ontology, manufacturing processes occur when the flow of parts and the flow of resources intersect at a unit process.

Daewoo uses a subset of the AARIA ontology. Task agents (corresponding to sets of unit processes) represent individual work orders, and resource agents

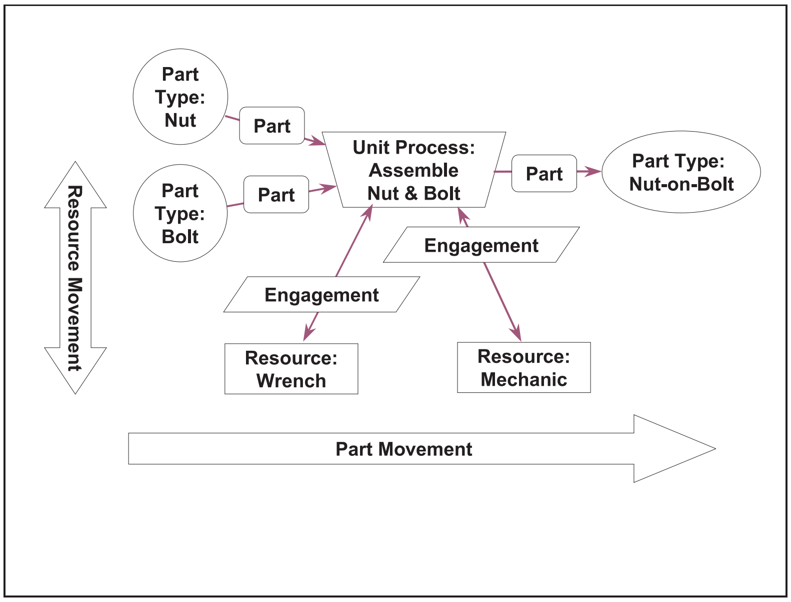

Figure 9.6 AARIA agents represent entities in the shop.

represent manufacturing resources such as machines. These domain-oriented agents are clustered into communities, and each community has several service agents: a bidding agent that handles all transactions among domain agents, a constraint propagation agent that propagates task dependencies and does some constraint satisfaction, and a meta agent that registers the skills of the domain agents in the community.

Agent Modeling: Critical information in manufacturing is usually organized by physical entities. Agents that represent these entities are the natural locus for maintaining this information. At Daewoo a resource agent caches information regarding previous bidding and the utilization of other compatible machines to guide subsequent bidding in directions that maximize overall goals and minimize later backtracking. In AARIA, resource agents also store maintenance and reliability information, while part type agents model the supply and demand of their parts over time. At Daewoo, each domain agent has a friend module in which it caches information about its colleagues that it obtains in the course of interaction. Each agent also has access to information about its community indirectly through the meta agent, and directly through a community-wide blackboard. The community information includes both present state and future objectives.

Agent Structure: All agents of a given class are the same. There are some shared modules (e.g., the friend module to store information about acquaintances). Agents do not change as they run, but do maintain state information.

Population: The Daewoo application has 30 machine agents and 700 task agents, together with the community's three service agents. Machine agents join the community when they are on-line and leave it when they go off-line. Task agents join the community when they are released to the shop, and leave it when they are completed.

Communication Channels: Agents communicate with one another electronically in two ways. The meta-agent provides a publish and subscribe service that agents can use to identify potential collaborators. Once one agent knows the identity of another, it communicates directly on the basis of information it has cached in its friend module. At Daewoo, messages do not persist, and agents do not travel. AARIA does use traveling agents to model parts as they move between part type agents and unit processes, and engagements as they move between resources and unit processes.

Communication Protocols: Daewoo uses a contract net negotiation protocol.

Configuration: Relations among agents are defined dynamically as a result of negotiation.

Coordination: The bidding process propagates constraints among the agents. Market interactions among agents generate a dissipative flow of currency to which the agents orient themselves, making efficient use of scarce resources.

Maturity: This system is in production use at Daewoo Motors in Korea.

9.4.4 Agents in Real-Time Control

Control systems operate faster and with more semantically constrained information than do planning and scheduling systems. They must provide real-time response, which is discussed further in Chapter 11.

Current technology for industrial process control offers many examples of coordinated pro-active objects that can usefully be viewed as agent-based systems. A typical chemical plant contains hundreds of PID control loops. PID stands for "proportional, integral, derivative," and describes the three functions that the agent can apply to the stream of data from a sensor. Each loop is a separate computer that adjusts some actuator as a function of various sensors. Because the action of one such loop changes physical quantities that will affect the behavior of other loops in the system, these loops can be viewed as (analog) agents that communicate through a shared environment. Table 9.4 summarizes six control systems that use digital agents.

The Zone Logic system applies agents to real-time control. Complicated manufactured parts such as engine blocks are often manufactured on a machine called a transfer line. Such a machine moves workpieces sequentially through a series of stations. At each position, individual mechanisms perform some specific task. For

System	Domain	What is an Agent?	Noteworthy Technology
ADS [43, 55, 56]	Steel Production	Group of related functions on a single processor	Subject-Based Addressing of messages; Hot swapping of running agents
AMROSE [65]	Robotic Control in Shipbuilding	One link in a segmented robot arm	Exploitation of locality of inter-agent communications
ARCHON [46, 49]	Electrical power grids [17]; Particle Accelerators [79]	Pre-existing expert system	Modeling partner agents
GM Paint Shop [23, 57]	Automotive paint shop control	Individual devices in the paint system (humidifiers, burners, chillers, steam)	Communication through the environment
Market-Based Climate Control [17, 18]	Building climate control	Office thermostats; air duct dampers	Market-based resource allocation
Zone Logic [81]	Handling and machining of complex mechanical parts	A single mechanism in a transfer line	Communication through the environment

Table 9.4 Real-time control systems.

example, the first station might bore a hole, the second might thread the hole, and the third might screw a hardened insert into the threaded hole.

A transfer line permits higher processing rates than discrete machines served by separate material transport systems. It contains a large number of mechanisms that must be controlled and coordinated. A typical transfer line may be a hundred meters long and contain dozens of stations with hundreds of mechanisms and more than 1500 degrees of freedom in movement overall. Traditional control schemes for such systems require the software engineer to understand the relations among all these mechanisms. When the system fails, identifying the responsible mechanism and the reason for the failure can be very time consuming. As a result, transfer lines often are down for maintenance more than half of the time. When the system is restarted after a failure, the various stations must be reset to a standard initial state, often requiring the scrapping or manual reprocessing of parts in process at the time of the failure. Because of the complex interactions among their mechanisms, transfer lines are notoriously difficult to keep operating. In many environments, 50% productivity is the most that can be achieved. By giving mechanisms autonomy, a

		State Variables						
Zone # & Name		**A**	**B**	**C**	**D**	**E**	**F**	**G**
1	**Initializing**	X	X	0	0	0.1		2, 5, 8
2	**Returned**	0	1	0	0			3
3	**Advancing from Returned**	0	1	1	0	1.0	9	4, 2
4	**Advancing Between**	0	0	1	0	5.0	9	5, 8
5	**Advanced**	1	0	0	0			6
6	**Returning from Advanced**	1	0	0	1	1.0	9	7, 5
7	**Returned Between**	0	0	0	1	5.0	9	4, 8
8	**Stopped Between**	0	0	0	0			4, 7
9	**Error Default**	X	X	0	0			1

0 = off; 1 = on; X = ignore
A = Advanced Limit Switch
B = Returned Limit Switch
C = Advance Motor Power
D = Return Motor Power
E = Max. Time in Zone (sec.)
F = Next Zone on Time-Out
G = Allowed Next Zones

Table 9.5 Rules for Zone Logic slide agent.

Zone Logic-controlled machine can readily achieve 90% productivity.

Agent Mapping: Zone Logic makes each mechanism in the transfer line (e.g., clamp, slide, transfer bar, probe) an agent that expects a certain range of conditions and knows what to do in each. This same mapping of agents onto basic physical entities in the control system is a common strategy for agent-based control, and appears also in AMROSE (where each link in a segmented robot arm is an agent), the GM Paint Booth system, and Xerox PARC's market-based climate control. These fine-grained agents that represent physical entities contrast with coarse-grained functional agents in the ARCHON and ADS systems. ADS is an older system, and probably owes its functional orientation to traditional methods of software design. ARCHON's functional structure reflects its mission to integrate pre-existing functional expert systems. In both cases, the functional approach reflects the transition from traditional software architectures, while physical decomposition seems to be the most direct approach to a new design.

Agent Modeling and Structure: Each mechanism maintains a rule base listing the state conditions it recognizes and what action it should take in each case, illustrated in Table 9.5. There are no explicit models of other agents and no explicit goals. For example, a slide mechanism includes two power switches (one to energize the motor that advances the slide, another to energize the motor that returns it) and two limit switches (one at the fully advanced position, the other at the fully returned position). The slide's state space thus includes $2^4 = 16$ possible states, only nine of which are physically possible. The other seven states reflect an error condition. For example, if both motors are energized or both limit switches are on, the mechanism has entered an error state and requires attention from an operator. For each allowed state or "zone," the mechanism's rule table lists the maximum amount of time the mechanism can spend in that zone and the allowable next zones it can enter. If the mechanism exhausts its time in a zone or if one of its state variables changes, it searches the list of next zones for one that it can enter by changing an output variable (e.g., actuating a motor), and makes the transition. If no specified next zone is accessible, it enters the error state and requests operator attention.

For example, if the slide is Advancing from Returned (state 3), the Returned limit switch should go off within one second. If it does not, the mechanism enters the error state and turns off both motors. Otherwise, it should next be found either in state 4 (Advancing Between) or state 2 (Returned), in case the advance motor shuts off before the slide leaves the Returned limit switch), and any other state defaults to Error.

Population: Each mechanism has its own agent, so a typical transfer line consists of hundreds of agents.

Communication Channels: Zone Logic agents communicate both physically and electronically. Individual mechanisms use physical sensors to determine the state and location of the part, thus adapting their behavior to what did or did not happen at earlier stations. Point-to-point non-persistent electronic communication between mechanisms guards against interference between mechanisms that may need access to the same physical space. The preferred design for agent-based control is to maximize interaction through the physical environment and minimize such explicitly coded dependencies between mechanisms, because explicit linkages make systems susceptible to failure when one mechanism is modified. Zone Logic agents are assigned to specific physical mechanisms installed at fixed locations on the line, and so do not need to migrate over a network.

Communication Protocols: Agent interaction in Zone Logic is directive. Both sensor information and interference signals are conditions in agent rules that lead reactively to action. Reactive protocols are especially well suited to low-level control environments, in which the digital logic must keep pace with physical events in the real world. At higher levels of control, exemplified by ARCHON, more complex protocols are useful.

Configuration: Agents are assigned to mechanisms when the transfer line is constructed. Which agents are active on a given part depends on an electronic processing file that accompanies the part through the system.

Coordination: one Logic agents coordinate their activity by propagating constraints. Market mechanisms are another candidate for real-time control, as seen in the Xerox market-based climate-control system.

Maturity: Zone Logic is deployed as a commercial product in several automotive manufacturing facilities.

9.5 How Does Industry Constrain the Life Cycle of an Agent-Based System?

The industrial life cycle poses restrictions and constraints on developing an agent-based system that are not present in most research environments. The cases in this section deal more with the tools and techniques used in constructing agent-based systems and less with the characteristics of the agent-based systems themselves.

The role of tools and methods defines an important distinction between industrial and academic projects. Academic laboratories often construct their own tools and methods, for two reasons.

1. Tools and methods may not exist to meet the challenges they explore.
2. Their educational mission is advanced by having students design and build tools.

In an industrial setting, a technology without supporting tools and methods has little hope of deployment, again for two reasons.

1. An industrial system is a means to an end, not an end in itself, and will be approved only if the firm can estimate its cost in advance and justify that cost against expected benefits. Well-defined tools and methods are the cornerstone of such a cost justification exercise.
2. The designers and implementers of industrial systems are first of all experts in the problems these systems are intended to solve, not in agent-based technology, and rely on tools and methods that package best practice in a way that they can use without becoming agent experts.

9.5.1 Requirements, Positioning, and Specification

The classical view of these phases of the life cycle is that they concern only *why* a system is needed and *what* it must do, not *how* its objectives are accomplished. On this view, the fact that the system will be implemented with agents is irrelevant, and firms should feel comfortable using traditional techniques for these phases. Thus little thought has been given to agent-specific mechanisms. Two caveats are in order.

1. Agents permit application of computers to highly-distributed, ill-structured problems that previously would not have been candidates for automation. Requirements can now be drafted for problems that would not have been addressed before. Engineers need to understand the benefits of agent-based systems over centralized monolithic systems, at least at a high level, to appreciate what kinds of problems such systems can address.
2. Agents establish a new paradigm for human-computer interaction that is less like the traditional master-slave relationship and more like a partnership. As

a result, the kinds of system-level behaviors that need to be specified will look more like specifications for a business process among people than does a traditional information system specification.

System behavior is one issue that needs to be determined in the requirements and specification phase. Others include interface constraints, performance constraints, operating constraints, life-cycle constraints (e.g., maintainability), economic constraints, and political constraints [82]. A complete design method for agent-based systems must take account of all these issues. This chapter focuses on the requirements that concern the behavior of the system.

At a high level, desired system behavior may be of several kinds. The system may maintain some set of state variables in a specified relationship with one another, thus exhibiting *homeostasis*. The system may be a *transducer* that converts specified stimuli into corresponding responses. Or the system may *learn* over time in response to its experience. At least two criteria are involved in a good behavioral specification.

- It should be specific enough to know if it has been achieved. A qualitative specification is usually adequate for role-playing, but we need a quantitative one to support simulation. For example, in a process control environment, "homeostasis" by itself is too vague. "Balance temperature and pressure" is OK for role-playing. Simulation requires specifying the quantitative link desired between pressure and temperature.
- It should be amenable to solution by architectural decisions. For example, the behavior "Have tooling available when needed" might be satisfied better by buying more tools than by expecting magic from agents. A better specification might be "Get high-value parts through the system first," "Identify relative scarcity of tool types," or "Reduce overall tool idleness."

The design team needs a concise statement of the problem to be solved and the constraints that must be observed. For example:

- What is the desired overall system behavior?
- What can be varied in the effort to achieve this behavior?
- What must not be touched?
- What approach is currently taken to solving the problem?
- Why is a new solution being contemplated? (Are there obvious shortcomings of the current solution? Is a change needed that is beyond the scope of the current solution?)

These questions are not exhaustive, but illustrate the kind of information that the requirements and specification process should produce.

The Agent Community (Social Level)	Protocols (dynamics of communication and coordination)
	Organization (roles or services of each agent with respect to the others)
The Individual Agent (Knowledge Level)	Local Planning (capabilities and plans)
	Local Behavior (reactivity, routine tasks)
	Local Knowledge (The agent's beliefs)

Table 9.6 What needs to be designed?

9.5.2 Design: The Conceptual Context

Design of an engineered artifact (such as an agent-based system) is a *process* that takes place within a conceptual *context*. In the agent research community, the "conceptual context" is often called an "agent architecture," and this subject has received considerable attention. Relatively less attention has been paid to the important question of the processes that designers go through. Industrial users will use agents more readily if basic principles and guidelines are available in both areas.

There is growing agreement among agent researchers on the set of issues that need to be resolved in order to design an agent-based system. Chapters 1 and 2 summarize a number of these. The common insight of all these proposals is that design must address both the individual agent and the community of which it is a part. Table 9.6 summarizes the various subcategories distinguished by one or another of these approaches.

9.5.3 Design: The Process

An iterative refinement approach is useful in designing agent-based systems [75]. The four stages outlined in Table 9.7 lead from a rough initial sketch of the community and its interactions to the point that software engineers can begin implementation. The stages are not strictly linear. Role-playing may show the need to rethink what agents are needed and what they should do individually; formal analysis may uncover a need for a revised organizational structure that requires more role-playing; and implementation design may raise further questions that require additional simulation. Still, there is a rough time ordering of these activities, in that conceptual analysis is the first to begin and implementation design is the last to complete.

Conceptual Analysis

The specification phase has defined what the system as a whole will do. Conceptual analysis gives an initial vision of what agents will be involved and how they will behave.

One widely-used technique for identifying objects in an object-oriented systems

Stage	Focus	Supporting Analysis	Answers
Conceptual Analysis	Components	Case Grammar	What system-level behavior is needed?
			What kinds of agents might we need to get it?
			How should they behave?
Role-Playing	Architecture (Kinematics)	Speech Acts & Dooley Graphs	How do the proposed agents interact with one another in an organization?
			What low-level behaviors are needed?
Formal Analysis	Behavior (Dynamics)	Formal Modeling , Simulation & Nonlinear mathematics	Are the descriptions logically consistent and complete?
			What kind of behavior emerges from realistic numbers of agents and interchanges?
Implemen-tation Design	Platforms & Tools	Simulation	How can this design become a deployable system?

Table 9.7 Stages in designing multiagent systems.

analysis [83] is to extract the nouns from a narrative description of the desired system behavior. A refinement of this approach is based on linguistic case theory [16, 27, 68]. Each verb has a set of named slots that can be filled by other items, typically nouns. Each slot describes the semantic role of its filler with respect to the verb. Thus the case role of a noun captures basic behavioral differences among entities in the domain, and is a candidate for an agent class.

To complete the preliminary decomposition, these categories are reviewed and possibly revised against overall system requirements and general principles of agent-based systems. Naturally occurring agent systems have proven remarkably robust and adaptable, and suggest a set of useful engineering principles [71].

Thing vs. Function. This chapter has repeatedly emphasized the natural precedents and practical benefits of physical rather than functional decomposition in agent-based systems. In most cases, deriving agents from the nouns in a narrative description of the problem to be solved yields things rather than functions. Legacy systems and watchdogs (agents that monitor the overall system for emergent behaviors) are two exceptions to this principle.

Small in Size. Natural systems like insect colonies and market economies are characterized by many agents, each small in comparison with the whole system. Such agents are easier to construct and understand than large ones, and the impact of the failure of any single agent will be minimal. In addition, a large population of agents gives the system a richer overall space of possible behaviors. (Very roughly, system state space is exponential in the number of agents.) The same benefits apply to artificial systems. Keeping agents small often means favoring specialized agents over more general ones, using appropriate aggregation techniques. For example, rather than writing a single agent to represent a complete manufacturing cell, consider an agent for each mechanism in the cell (e.g., one for the fixture, one for the tool, one for the load-unload mechanism, one for the gauging station).

Decentralized. Natural systems are not centralized as artificial systems often are. There are several reasons for imitating this tendency. A central agent is a single point of failure that makes the system vulnerable to accident. It can easily become a performance bottleneck. More subtly, it tends to attract functionality and code as the system develops, pulling the design away from the benefits of agents and regressing to a large software artifact that is difficult to understand and maintain. Centralization can sometimes creep in when designers confuse a class of agents with individual agents. For example, one might be tempted to represent a bank of paint booths as "the paint agent," because "they all do the same thing." Certainly, one would develop a single class for paint-booth agents, but each paint booth should be a separate instance of that class.

Diversity and Generalization. Natural communities of agents balance diversity (which enables them to monitor an environment much larger than any single agent) with generalized mechanisms (enhancing their interaction with one another and reducing the need for task-specific processing). For example, pheromones enable insects not only to map out paths to food sources, but also to coordinate nest construction. Conventional class inheritance mechanisms support generalization across agents, but the hard part is identifying appropriate generalizations in the first place. Early designs typically multiply differences among agents, while later refinements make more effective use of inheritance.

With a candidate set of agents in hand, the next step is to hypothesize their individual behaviors and the classes of messages they can exchange, keeping in mind the desired overall system behavior. This process is intuitive, not algorithmic. Some behaviors may be obvious, but there will always be subsystems where only simulation of example agent behaviors (first in role-playing, later on a computer) can verify the right behaviors. At this point, the main concern is to identify the decisions each agent needs to make and the other agents with which it needs to make them rather than on the details of each agent's internal reasoning. Again, principles observed in naturally occurring systems help evaluate candidate agent dynamics and interactions.

Concurrent Planning and Execution. Traditional systems alternate planning and execution. For example, a firm develops a schedule each night for its manufacturing operations the next day. The real world tends to change in ways that invalidate advance plans. Natural systems do not plan in advance, but adjust their operations on a time scale comparable to the environment's rate of change. Watch out for behaviors that involve extensive up-front planning.

Currency. Naturally occurring multiagent systems often use a flow field, such as the flow of money in a market economy and the evaporation of pheromones in insect communities. These mechanisms accomplish two purposes. They provide an "entropy leak" that permits self-organization (reduction of entropy) at the macro level without violating the second law of thermodynamics overall, and they generate a gradient field that agents perceive and reinforce and to which they can orient their actions, thus becoming more organized [51]. Wherever possible, artificial agent communities should include such a flow.

Local Communication. Agents need to limit the recipients of their messages [86]. Wherever possible, instead of "broadcast X," define more precisely the audience that needs to receive the message.

Information Sharing. Agents often need to share information across both time and space. ("Learning" thus becomes a special case of information sharing.) Phylogenetic learning is not nearly as demanding as the ontogenetic mechanisms developed in classical AI, and sociogenetic mechanisms can be even simpler

Role-Playing

With agents identified and tentative behaviors described, the emergent behavior of selected subsystems can be explored by having people play the roles of the various agents. Such a rehearsal does not show the full dynamic behavior expected from a complete population of agents operating at computer speed, but does validate the basic behaviors needed and provides a basis for defining some internal details of computerized agents. Where computer agents supplement the activity of human operators, the role-playing exercise also helps capture the techniques, knowledge, and rules that the humans have been using to ensure that the computer agent augments this behavior appropriately [8].

Many of the individual behaviors for most of the agents will be fairly obvious. This situation is fortunate, since role-playing a complete system as small as 50 or 100 agents can be slow, tedious, and inconclusive. To explore the emergent behaviors of the system in regions that are not obvious, analysis focuses on subsystems of a dozen or so agents where there are significant questions about the match between individual and system behaviors.

Role-playing requires both identified subsystems and several scripts of the desired system behavior. For example, role-playing a system with homeostasis requires a list of the state variables that can independently change, the range of variation that they can expect, and the corresponding corrections needed in other variables. These scripts guide the role-playing activities. Because of the time and effort constraints of role-playing, they will sample the overall space of desired system behaviors only sparsely, and should be chosen to explore widely separated regions of this space.

A separate person should represent each agent in the subsystem identified in the conceptual analysis phase. When there are many more agents than people available, a single person may handle a complete class of agents. In this case one must distinguish carefully between the behavior of the agent class and what a single agent of that class can know. Agents, even those of the same class, do not have direct access to one another's variables, and people representing them in a role-play need to be careful not to "leak" information among them.

The environment in which agents live is not necessarily passive, but may have state and processes associated with it. In addition to the agents proposed for the system being engineered (the "system agents"), someone should play the role of the environment. This person raises external conditions as called for in the script, receives actuator outputs from the system agents, and integrates these outputs into their overall effect on the environment, thus monitoring the system's ability to achieve the required changes. The facilitator can represent a simple environment. When the environment is more complicated, its representative may need to do more extensive reasoning, and should be separate from the facilitator.

The primary responsibility of participants in role-playing is to figure out the rules that should guide the behavior of the agent for which they are responsible. The structure of the conversation among agents will emerge naturally from the interaction, and can be retrieved by post-hoc analysis, but the internal rules need to be developed by the participants themselves. A stochastic process (such as rolling a die or flipping a coin) is used to choose among internal agent decisions that later will be the subject of detailed computation. (By treating a penny, a dime, a nickel, and a quarter as successively higher bit positions, up to 2^4 alternatives can be represented.)

All actions among role-players are recorded for later analysis. These actions may be either speech acts (messages to other agents) or non-speech acts (influences on the environment). The agents record these actions on cards that are then given to the participant representing the receiving agent (for a message) or the environment (for a physical action). Each card records five pieces of information, in addition to the actual content of the message:

1. The identity of the sending agent
2. The identity of the receiving agent
3. The time the card is sent
4. The identity of the agent whose card stimulated this one
5. The time that the card stimulating this one was sent

This information enables reconstruction of the thread of conversation among the agents. The time entries capture the order in which messages are generated. Ideally, one could assign a unique sequence number to all cards, but the task of maintaining such a number across all participants is burdensome and prone to error. By placing a digital desk clock in full view, it is easy to maintain an unambiguous ordering of the cards.

A facilitator who is not one of the agents oversees the execution of each script. There are three phases in this responsibility.

1. *Initiate:* The facilitator announces that a new script is starting. If the facilitator and environment are not the same person, the facilitator makes sure the correct script drives the environment.
2. *Run:* While the participants are running the script, the facilitator carries message cards between agents, watches for possible cross-talk ("Isn't your action based partly on what B said a few moments ago to C? Should you have been included on the distribution for that message?"), and (if also serving as the environment) simulates exogenous inputs to the system and accounts for the effect of outputs.
3. *Debrief:* After completing a script, the facilitator helps participants synthesize important conclusions from the session. For example: What operational decisions could not be resolved locally? What state information does each agent need to maintain? How complex do agents need to be? Are participants conscious of internal state shifts?

Enhanced Dooley Graphs [69] can help analyze conversations in agent-based systems. Each node in the graph represents an agent in a role. A given agent may appear at different nodes if it changes roles in the course of the conversation. These roles suggest units of behavior that can often be reused across an agent community. Thus they provide a first-level decomposition of individual agents into behaviors, and guide the initial coding of the system.

Formal Analysis

Brainstorming and role-playing are flexible, creative ways to explore possible agent designs, but their results need to be checked more formally before implementation begins. Two important tools are formal modeling and simulation. With a rudimentary design in place, a logician can develop a formal model of the individual agents and their interactions over time. Logical manipulation of this model can then test for consistency and completeness against project requirements. The research program of the DESIRE team [22] is one example among many of this approach. Like the "correctness proof" approach to program development, formal analysis of an agent design is complex and expensive, and should be used selectively. Simulation is a more broadly used tool, and a more necessary one. It enables the designer to observe and evaluate the emergent behavior of the entire community, and to test how the behavior seen in a role-play scales up to a full population. The growing acceptance of genetic methods in industry opens the door for using simulation to grow agents, avoiding the need to program them manually. The code of the simulated agents can serve as a detailed design for the final implementation.

Implementation Design

In preparation for implementation, the designer selects the deployment platforms and tools that will be used in the fielded system. Sometimes these choices are known at the outset. In other cases, the results of the earlier steps of design may guide implementation design, as when simulation studies show that the required level of performance requires agents to execute on separate processors.

9.5.4 System Implementation

The various tools described in this section vary widely in their functionality and capabilities. Often the tool needed for a given application will depend on the details of the design that has been developed using the methods of the last section.

Hardware

General-purpose computers dominate agent research, but many industrial applications are better served by parallel architectures that can assign a single processor to each agent. Such an architecture supports real-time control applications much better than do general-purpose operating systems such as UNIX or Windows, and most of the examples here permit processors to be distributed physically so that software agents can be embedded in physical devices.

The simplest products in this category are single-chip or single-board microcomputers, such as the Basic Stamp [66] or Z-World's C-based offerings [94], with a dozen or so I/O points and a supporting PC-based development environment. Such platforms require additional peripheral support for inter-agent communication over any significant distance. The next level of sophistication is represented by LonWorks [21], which is built around a single-chip computer that includes LonTalk, a complete 7-layer OSI protocol. The product line includes a wide variety of transceivers for a variety of interconnections among individual chips, and interfaces to other networks.

An example of the high end of dedicated agent hardware for industrial applications is the Flavors PIM (Parallel Inference Machine) [25], a centralized parallel computer that is available either as a board for a Macintosh with four powerPC604e processors, or in a VME format with up to 125 68040 processors. Each processor supports 125 virtual "cells" or virtual processors, in a real-time operating environment.

Standards

Broadly accepted standards bring users and developers together into a critical mass. If the requirements of various users differ widely from one another, developers will not have a large enough market for any single technology to justify the expense of bringing it to commercial status. If the offerings of different developers do not work together, users will not be able to assemble the full suite of tools that they require. To the extent that agent standards agree with standards currently deployed in the pre-agent environment, they enable incremental introduction of agents, an approach that is less painful and more likely to be accepted by management than requiring a wholesale redesign of the factory to accommodate agents.

Two organizations are devoted specifically to the definition and promulgation of standards for agent-based systems. The National Industrial Information Infrastructure Protocols program (NIIIP) [59] is a consortium of U.S. companies addressing the problem of enabling manufacturers and their suppliers to interoperate as effectively as if they were part of the same enterprise. FIPA [13], the Foundation for Intelligent Physical Agents, is a world-wide consortium devoted to agent standards in general.

Commercial developers of agent tools draw on a wide variety of standards for distributed systems and networking. The following examples begin with the lowest levels of agent communication and extending to the high-level definition of agent behavior.

Physically moving bytes around can be a problem in an industrial environment. Electromagnetic interference from arc welders, induction furnaces, and motor starters can overwhelm many network structures that are completely adequate for laboratory work. Until recently, industrial control relied on point-to-point wiring of I/O between a controller and the devices being controlled, with separate conductors for each device. Control-area networks such as DeviceNet [63] replace these unwieldy tangles with multiplexed communications, and are engineered to cope with potential interference.

Once an electronic pathway exists between agents, they need to be able to find one another. Heterogeneous platforms are the rule rather than the exception in many factories: a shop-floor server running on a DEC machine may support machine controllers from Fanuc, Rockwell Automation, and Modicon, and a number of applications built on an industrially-hardened personal computer platform. CORBA [64], the Common Object Request Broker Architecture, defines a standard mechanism by which objects written in different languages and executing in a distributed environment can make requests of, and respond to, one another.

After finding one another, agents need to be able to exchange information and express themselves about it. Chapter 2 discusses in detail two important standards that support inter-agent communication: KIF [34], the Knowledge Interchange Format, and. KQML [28], the Knowledge Query and Manipulation Language. KIF expresses the content of a proposition, while KQML expresses the agent's attitude toward the proposition.

Sometimes it is not enough for agents to talk to one another over the network. If their interactions are intensive, they should share the same processor. A part agent may need to move from one machine agent to another during its residency in the shop. Java [85] enables agent behavior to travel from one processor to another, and thus provides a way for agents themselves to travel over networks and execute on diverse platforms.

These standards provide interoperability between different computer systems. Another category of standards enables people to communicate effectively with agents. Industrial engineers have evolved their own conventions for specifying and implementing systems, and they will accept agent technologies more readily if an agent system supports these conventions. For example, Grafcet [9] is an international standard (IEC 848) for a graphical control language based on Petri nets. Petri nets are a powerful mechanism for representing the internal logic of an agent [26], and Grafcet enjoys widespread industrial use as a representation for control logic, making it a good candidate for implementing industrial agents.

Tools

A growing array of software tools is available for developers of agent-based systems. Section 9.6 discusses several examples.

9.5.5 System Operation

Little specific case information is available at this time concerning the operation of industrial agent-based systems, but the nature of agent technology suggests two important issues that will make the difference between successful and unsuccessful systems.

Agent Dynamics. One of the great benefits of agent-based systems is their ability to generate complex system-level performance from relatively simple individual agents. This system-level behavior often cannot be predicted analytically from the descriptions of individual agents, but must be observed in simulation or real-life. As a result, the detailed behavior of an implemented system may not be known in advance, and individual agent behaviors may need to be modified in real time as the system runs. The tools to support the monitoring, analysis, and adjustment of an agent-based system in operation are the same ones needed to design the system in the first place. Thus one expects that the more successful development tools discussed in the previous section will take on more and more features of operational interfaces, such as simplicity for use by non-programmers, alarm and emergency management, and data logging and archiving.

Humans and Agents. Agent-based systems require closer interaction between human and computer than do traditional systems, even as they enable automation of many tasks that previously required human attention. The reason for this paradox is that the autonomy of agents, one of their main strengths, moves them from a position of an obedient slave to that of a cooperating partner. One can tolerate bad manners on the part of a slave, but people who relinquish decision-making ability to a silicon peer expect a certain level of etiquette on the part of their new associates. The problem is more complex because people can learn to recognize their own bad habits and modify their behavior accordingly, but at the present state of technology, acceptable demeanor must be programmed into computer agents. Successful operations requires systems that embody not only advanced computational science but also sophisticated psychological understanding of how people work together and what makes teams successful.

9.6 Development Tools

Development tools are one of the most powerful ways to move a new technology into widespread use. Object-oriented programming was possible before the advent of specific object-oriented languages, but each team had to define its own conventions and rely on the expertise of individual developers to enforce them. Wide-spread industrial acceptance came only with languages such as Smalltalk, C++, and Objective C that package an agent model and enforce a set of best practices about how to use it. The lack of commercially supported development environments has been a major roadblock to wider agent deployment in industry [70].

Several products have recently emerged to address this need. Since one function of a tool is to enforce best practice, it is not surprising that each of these tools emphasizes its own view of what an agent is and what kind of resources agent developers need. This section groups example tools under five such views, ordered roughly from simplest to most complex. Unless otherwise noted, the tools described in this section are commercially available and supported. Chapter 1 discusses several tools that are more oriented toward research.

An Agent as a Single Reactive Process. IBM's Agent Building Environment [41] is an extensible C++ library for constructing rule-based forward chaining agents, and is designed for applications such as monitoring a network information source and informing a human when certain conditions are satisfied. ABE represents its facts and rules in KIF. The core reasoning engine can be attached to procedures external to itself. These procedures, which can be written in C++ or Java, provide the means of sensing conditions in the environment, taking action in the environment, and triggering inferencing activity in the underlying engine. Attached procedures are packaged into "adapters," and the ABE distribution includes a number of predefined examples: an alarm clock that can trigger time-based events, a USENET News monitor, an HTTP monitor, an adapter that can observe and manipulate files, an Email sender, and an example of an adapter that fetches stock quotes from a WWW site. ABE is designed to support isolated intelligent agents rather than multiagent systems.

Agents as Capitalists. Dissipative mechanisms such as currency flows are a powerful way to coordinate a decentralized system. Agorics is a software development company that applies market mechanisms to real-world problems. Agorics is developing Joule, a programming language for distributed concurrent asynchronous systems, that supports market-based computing with the encapsulation of resources and the management of access to them [1]. Joule raises the communication channel between agents to first-class status as a "channel," a unidirectional route with two "ports." Agents (called "servers" in Joule) place messages into a channel's acceptor port and read them from a distributor port. These ports can be passed from one agent to another, thus controlling access to the services provided by a given server. The message objects themselves handle authentication and other security concerns. Currently, Joule is not distributed externally, but is used within Agorics on industrial projects for its clients.

Agents as Travelers. Some agents can migrate from one processor to another. This capability permits agents that must conduct a high-bandwidth conversation to move to a common processor so that the network as a whole is not burdened with the traffic between them. It also permits local communities of agents to interact with one another even when the processor on which they are located is disconnected from the rest of the network. The earliest widely publicized tool for mobile agents is General Magic's proprietary Telescript language [90-92], which models the world as places (processors) and agents (processes). Several firms are implementing these concepts in Java to avoid the need for a proprietary language interpreter on each processor that an agent might wish to visit. IBM's Aglets [42] are Java objects that can move from one host to another. The Java Aglet API (J-AAPI) defines the methods necessary to create aglets, handle messages, and manage the course of the aglet's life. Danny Lange, the lead developer of Aglets, has joined General Magic, and is now a member of the Odyssey team that is producing a Java version of Telescript [33]. ObjectSpace's Voyager [62] provides a Java-based Object Request Broker (ORB) designed for mobile agents. ObjectSpace offers a detailed comparison of Aglets, Odyssey, and Voyager [61]. All three tools are available free for noncommercial use, and Voyager is freely available for commercial use as well.

Agents as Members of a Community. The next level of sophistication in agent tools provides explicit support at the level of the community, with special emphasis on communication mechanisms. As functionality is added to the Java-based frameworks discussed in the previous section, they will come to look more like these tools as well.

Gensym's Agent Development Environment (ADE) [35] builds on its widely accepted G2 object-oriented environment, a robust real-time platform for industrial deployment of AI techniques. The ADE provides a predefined class hierarchy of agents and agent parts, agent communications "middleware," a graphical specification and programming language for agent behavior based on the Grafcet standard for industrial-strength Petri nets, and a simulation tool for performing simulations of (distributed) agent-based applications. ADE supports agents running on a distributed network of computers as well as on a single machine. Each agent has a network-wide unique identifier, and agents can be grouped into nested "environments," each of which is also an agent. ADE itself provides a basic direct addressing message service, to which users can add additional functionality (such as guaranteed delivery or subject-based addressing). An agent uses messages not only to communicate with other agents but also to queue up events for its own subsequent activity.

In support of the AARIA project described in Section 9.4.3, Intelligent Automation, Inc., the AARIA prime contractor, created the first version of Cybele, a NeXT-based agent infrastructure [24]. Based on the requirements analyzed in [52], Cybele supports agent creation and deployment over a network of varied platforms, a message addressing scheme for agent communication that is independent of the location of a sending or receiving agent, the accumulation of messages intended for a currently busy recipient agent, the proper conversion of message data across platforms, multicasting, broadcasting, and peer-to-peer messaging, and the migration of agents across processors for performance optimization and/or fault tolerance. Building on the lessons learned in the initial implementation, Cybele is currently being reimplemented in Java.

Metra's UNITY_Agent [54] is a Java-based agent environment that builds on the three-level "tactical, operational, strategic" agent hierarchy of [10]. Agents are grouped into communities, themselves agents, each with a Meta Agent that monitors the identity and capabilities of agents in the community. The framework supports communication via direct addressing, subject-based addressing, and blackboards. The base agent class includes capabilities for modeling self and other agents, performing situation assessment, managing local tasks, negotiating for resources, and competitive bidding to resolve conflicts. Users configure a new agent by specifying the agent's attributes, defining trigger events, writing actions to be taken on a trigger event in Java, and binding trigger events and actions in rules. The interface to the external world is CORBA-based and supports standard database access. UNITY_Agent is the environment underlying the Daewoo scheduling system described in Section 9.4.3.

Agents as Intelligent Processes. Much agent research grows out of the Artificial Intelligence community, and assumes that individual agents aspire to some level of intelligence. The tools described thus far do not provide any explicit support for individually intelligent agents, and are appropriate for systems of relatively simply agents whose interactions produce intelligent system-level behavior. The tools in this section embody specific models of individual intelligence, and illustrate how a well-crafted tool can make complex techniques accessible to a wide range of users.

dMARS [36], a descendant of SRI's Procedural Reasoning System, is an instantiation of the BDI model of agents, according to which agents should explicitly model their Beliefs, Desires, and Intentions. Each dMARS agent includes a Belief Database of current beliefs about the world, a Goal Database of objectives or desires to be realized, a Plan Library of context-sensitive procedures that the agent can use to achieve goals and react to situations, an Intention Structure of tasks to be performed, and a Task Manager that repeatedly selects a Plan based on the agent's Beliefs and Goals, places it in the Intention Structure, and manages its execution. Plans can be conditioned either on external conditions or on the state of the agent's internal databases, and so can support reflective planning and dynamic reprogramming of the agent.

D-Muse [93] is a distributed version of an earlier commercial AI toolkit, MUSE. Based on the PopTalk object language, MUSE offers a complete frame representation system with forward and backward chaining that supports real-time operation. D-Muse provides a layered set of communication capabilities that enable the interaction of individual MUSE-based agents. Agents interact mainly through mirrored objects. One agent (the publisher) creates a master object that is then made visible to one or more subscriber agents by being copied as a slave object within the subscriber. Whenever the publisher modifies the master object, D-Muse synchronizes the slaves. A subscriber cannot modify a slave object, but can attach demons or relations to it, match rule patterns to it, or manipulate it in any other way that would be possible within the native MUSE environment.

The Agent Building Shell (ABS) [5, 7] at the University of Toronto is a set of object classes and supporting tools that implement a four-layer architecture for coarse-grained agents. The *knowledge management layer* provides general-purpose representation and inference mechanisms that agents can use to model their knowledge and beliefs about the problem domain, the environment (including other agents), and themselves. The *ontology layer* uses the knowledge management layer to construct the specific models that an agent maintains of its domain, its environment, and itself. The *cooperation and conflict layer* supports two services to manage shared knowledge between agents: a subscription-based information service that enables agents to be notified automatically when information of interest to them is posted, and mechanisms for managing an agent's beliefs when it receives contradictory information from other agents. The *coordination and communication layer* is supported by the Coordination Language (COOL) and provides inter-agent communication using a superset of KQML, definition of arbitrary inter-agent protocols, and integration of legacy applications. While the Agent Building Shell is not offered commercially, it embodies techniques of knowledge representation and automated inferencing that have been deployed commercially in expert system shells such as KnowledgeCraft and KEE, and shows how classical AI methods are migrating into agent tools.

9.7 Conclusions

This brief survey of industrial agent systems suggests two ways in which such systems differ from research systems: the systems must be practical, and the tools used to develop them must be packaged.

Industrial systems are driven by the need to solve a practical problem, rather than curiosity about the possibility of some technology. The criterion for success in an industrial project is not how clever the technology is, or what one has learned about that technology, but how well the system solves the problem that it addresses. The entire life cycle of an industrial system is shaped by this unrelenting pressure to make a difference to the firm's effectiveness. At first glance this focus on profit and practical results strikes some researchers as confining and unimaginative. In fact, the complexity of real-world problems offers intellectual challenges every bit as stimulating as the more theoretical challenges of the research laboratory, and the unforgiving nature of the business environment provides a much clearer sense of success or failure than can be achieved in a more abstract domain.

The methods of designing, building, operating, and maintaining agent-based systems must be packaged if they are ever to find wide-spread deployment in the industrial world. The orientation to practical problems means that engineers in industry must be first of all experts in the products they manufacture, the processes they control, or the services they render. Agent technology is for them a means to an end, a tool. The more the tool fades into the background and lets them concentrate on the requirements of the problem at hand, the more likely they are to use it.

The PAAM (Practical Application of Intelligent Agents and Multiagent Technology) conferences [88] are more oriented toward applications than other agent conferences, and their proceedings are a good source of further readings. As with other technologies, detailed application issues are more likely to be discussed in venues associated with the application domain than in those dedicated to the underlying technology, and as a result the best case studies will be scattered throughout a wide range of conference proceedings and journals. Ultimately, application expertise is best communicated by hands-on experience rather than by papers, and readers eager to learn more about this area should establish joint projects with industrial partners around application problems of industrial scope and complexity, where the objective is to improve the industrial partner's operations rather than to generate research reports.

The big open issue in applications of DAI is the instantiation of the techniques that researchers develop in standards and development tools that make them accessible to industrial users. The best techniques will not be widely used unless they are embedded in commercially supported tools. Now that multiple products are becoming available, market forces will join technical excellence in determining the platforms on which industry will build in the future. Researchers who are alert to these market forces and who pay special attention to packaging and deployment of their results will see their work have the most lasting impact on the field.

9.8 Exercises

1. *[Level 3]* The deployed agent-based systems described in this chapter focus on the design and operation phases of the life cycle. Why is this the case? Make friends with some industrialists, identify the main challenges in each phase of the life cycle of some process or product of interest to them, and see if you can propose an application for agents in some of the neglected phases.
2. *[Level 4]* Formulate a project proposal to do an agent application in one of these neglected phases, in partnership with your industrial acquaintance. Obtain funding for the project and execute it.
3. *[Level 2]* Analyze a recent or upcoming research project in your laboratory according to the life cycle pattern outlined in this chapter. To which steps do you usually devote attention? Which ones are novel or unusual in your environment? For each of these steps that is not part of your usual project cycle, identify the difference between an industrial setting and your own that makes it important to an industrial user.
4. Software packages that support an industrial team as they work their way through the life cycle are called "work flow" packages, and research concerning them is centered in the discipline of computer-supported collaborative work.
   (a) *[Level 1]* Conduct a literature review on workflow packages. Compare and contrast the ways different solutions decompose the problem.
   (b) *[Level 2]* Design an agent-based workflow program. Explain how your decomposition into agents supports the industrial requirements outlined in Section 9.2.
   (c) *[Level 3]* Implement your program.
   (d) *[Level 4]* Field-test your program with an industrial partner. If the test is successful, start a company to market it.
5. *[Level 1]* Compare the agent architecture used in your work with that discussed in Section 9.4.2 above. Which categories do you design explicitly, and which are left implicit? Are there other categories, not discussed here, which industrial agent designers should consider?
6. *[Level 2]* Itemize the development tools that you use in your projects. For each tool that is developed in your own laboratory, answer the following questions:
   - What philosophy of agents does this tool impose on your work?
   - What functionality does it provide that is not yet available in the market?
   - What functionality of more widely available tools does it duplicate?
   - How does the capability of this tool, and the availability (or lack of availability) of commercial sources for this capability, impact the prospect for transferring your discovers to industrial practice?

7. *[Level 4]* Implement your next project in a commercial development environment that meets as many of your requirements as possible. Explore ways to package the additional functionality you require as extensions to this environment. Transfer these extensions to the vendor of the development environment for them to make available in the next release of their product.

9.9 References

1. Agorics. *Joule: Distributed Application Foundations.* Agorics Technical Report ADd003.4P, http://www.webcom.com/~agorics/joule.html, Agorics, Inc., Los Altos, CA, 1995.
2. J. S. Albus, H. G. McCain, and R. Lumia. *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM).* NBS Technical Note 1235, National Bureau of Standards, Gaithersburg, MD, 1987.
3. A.D. Baker. *Manufacturing Control with a Market Driven Contract Net.* Ph.D. thesis, Rensselaer Polytechnic Institute, Electrical Engineering, 1991.
4. A. D. Baker, H. V. D. Parunak, and K. Erol. *Manufacturing over the Internet and into Your Living Room: Perspectives from the AARIA Project.* http://www.aaria.uc.edu/cybermfg.ps, Department of Electrical & Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, 1997.
5. M. Barbuceanu. The Agent Building Shell: Programming Cooperative Enterprise Agents. http://www.ie.utoronto.ca/EIL/ABS-page/ABS-intro.html, 1997.
6. M. Barbuceanu and M. S. Fox. The Architecture of an Agent Based Infrastructure for Agile Manufacturing. In *Proceedings of IJCAI-95*, 1995.
7. M. Barbuceanu and M. S. Fox. The Architecture of an Agent Building Shell. In *Proceedings of Agent Theories, Architectures, and Languages*, pages 235-250, Springer, 1995.
8. D. Bellin and S. S. Simone. *The CRC Card Book.* Addison-Wesley, 1997.
9. E. Bierel and J.-M. Roussel. "Welcome to the GRAFCET Home Page". http://www.lurpa.ens-cachan.fr/grafcet.html, 1995.
10. P. Burke and P. Prosser. The Distributed Asynchronous Scheduler. In M. B. Morgan, editor, *Intelligent Scheduling*, pages 309-339. Morgan Kaufman Publishers, Inc., San Francisco, 1994.
11. J. Butler and H. Ohtsubo. ADDYMS: Architecture for Distributed DYnamic Manufacturing Scheduling. In A. Famili, D. S. Nau, and S. H. Kim, editors, *Artificial Intelligence Applications in Manufacturing*, pages 199-214. AAAI Press/The MIT Press, Menlo Park, CA, 1992.
12. F.-C. Cheong. *Internet Agents: Spiders, Wanderers, Brokers, and Bots.* Indianapolis, IN, New Riders, 1996.
13. L. Chiariglione. Foundation for Intelligent Physical Agents. http://drogo.cselt.stet.it/fipa/, 1987.
14. K. T. Chung and C.-H. Wu. *Dynamic Scheduling with Intelligent Agents: An Application Note.* Metra Application Note 105, Metra, Palo Alto, CA, 1997.
15. S. H. Clearwater, editor. *Market-Based Control: A Paradigm for Distributed*

Resource Allocation. Singapore, World Scientific, 1996.

16. W. A. Cook. *Case Grammar: Development of the Matrix Model*. Washington, Georgetown University, 1979.
17. J. M. Corera, I. Laresgoiti, and N. R. Jennings. Using ARCHON, Part 2: Electricity Transportation Management. *IEEE Expert*, 11(6):71-79, 1996.
18. M. R. Cutkosky, R. S. Englemore, R. E. Fikes, T. R. Gruber, M. R. Genesereth, W. S. Mark, J. M. Tenenbaum, and J. C. Weber. PACT: An Experiment in Integrating Concurrent Engineering Systems. *IEEE Computer*, 26 (January)(1):28-37, 1993.
19. T. P. Darr and W. P. Birmingham. An Attribute-Space Representation and Algorithm for Concurrent Engineering. *AI EDAM*, 10(1):21-35, 1996.
20. N. A. Duffie, R. Chitturi, and J. I. Mou. Fault-tolerant Heterarchical Control of Heterogeneous Manufacturing System Entities. *Journal of Manufacturing Systems*, 7(4):315-28, 1988.
21. Echelon. Welcome to Echelon. http://www.lonworks.echelon.com, 1997.
22. P. v. Eck. The DESIRE Research Programme. http://www.cs.vu.nl/vakgroepen/ai/projects/desire/, 1996.
23. G. Ekberg. Benefits of Autonomous Agent Approach to Manufacturing Systems Control. In *Proceedings of Third Annual Chaos East Technical Conference*, R. Morley, Inc., 1997.
24. K. Erol. Cybele: An Infrastructure for Autonomous Agents. http://www.i-a-i.com/projects/cybele/index.html, 1996.
25. P. Fandel, R. DeSimone, and H. Mitchell. Paracell/PIM Product Summary Preface. http://www.flavors.com/docn/ppps/index.html, 1996.
26. J. Ferber. *Les systèmes multi-agents: vers une intelligence collective*. Paris, France, InterEditions, 1995.
27. C. J. Fillmore. The Case for Case Reopened. *Studies in Syntax and Semantics*, (8):59-81, 1977.
28. T. Finin. UMBC KQML Web. http://www.cs.umbc.edu/kqml/, 1997.
29. K. Fordyce, R. Dunki-Jacobs, B. Gerard, R. Sell, and G. Sullivan. Logistics Management System: An Advanced Decision Support System for the Fourth Decision Tier Dispatch or Short-Interval Scheduling. *Production and Operations Management*, 1(1):70-86, 1992.
30. K. Fordyce and G. G. Sullivan. Logistics Management System (LMS): Integrating Decision Technologies for Dispatch Scheduling in Semiconductor Manufacturing. In M. B. Morgan, editor, *Intelligent Scheduling*, pages 473-516. Morgan Kaufman Publishers, Inc., San Francisco, 1994.
31. M. S. Fox. The Integrated Supply Chain Management Project. http://www.ie.utoronto.ca/EIL/iscm-descr.html, 1996.
32. M. S. Fox, J. F. Chionglo, and M. Barbuceanu. The Integrated Supply Chain Management System. http://www.ie.utoronto.ca/EIL/public/iscm-intro.ps, Department of Industrial Engineering, University of Toronto, Toronto, Ontario, 1993.
33. General Magic. General Magic: Odyssey. http://www.genmagic.com/agents/odyssey.html, 1997.
34. M. R. Genesereth. Knowledge Interchange Format (KIF). http://logic.stanford.edu/kif/, 1996.

35. Gensym. Agent Development Environment (ADE) Overview. Gensym, Inc., Cambridge, MA, 1997.
36. M. Georgeff. dMARS Technical Overview. http://www.aaii.oz.au/proj/dmars_tech_overview/dMARS-1.html, 1996.
37. J. Hatvany. Intelligence and Cooperation in Heterarchic Manufacturing Systems. *Robotics & Computer-Integrated Manufacturing*, 2(2):101-104, 1985.
38. R. Hauser and D. Clausing. The House of Quality. *Harvard Business Review*, 66(May-June):63-73, 1988.
39. J. Heaton. *Agent Architecture Distributes Decisions for the Agile Manufacturer: Reengineering at AlliedSignal Automotive Safety Restraint Systems*. AMR Report, (June):8-13, 1994.
40. J. H. Holland. *Hidden Order: How Adaptation Builds Complexity*. Reading, MA, Addison-Wesley, 1995.
41. IBM. IBM Agent Building Environment (ABE) – A toolkit for building intelligent agent applications. http://www.networking.ibm.com/iag/iagsoft.htm, 1997.
42. IBM. IBM Aglets Workbench-Home Page. http://www.trl.ibm.co.jp/aglets/, 1997.
43. H. Ihara and K. Mori. Autonomous Decentralized Computer Control Systems. *IEEE Computer*, 17(8):57-66, 1984.
44. N. Jennings. Applying Agent Technology. Plenary presentation at PAAM'96. 1996.
45. N. R. Jennings. ADEPT: Advanced Decision Environment for Process Tasks. http://www.elec.qmw.ac.uk/dai/projects/adept/, 1997.
46. N. R. Jennings, J. Corera, and I. Laresgoiti. Developing Industrial Multi-Agent Systems. In *Proceedings of 1st Int. Conf. on Multi-Agent Systems*, pages 423-430, AAAI Press, 1995.
47. N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E. Wiegand. Agent-based business process management. *International Journal of Cooperative Information Systems, Forthcoming*, 1997. Available at ftp://ftp.elec.qmw.ac.uk/pub/isag/distributed-ai/publications/IJCIS96.ps.gz.
48. N. R. Jennings, P. Faratin, M. J. Johnson, P. O'Brien, and M. E. Wiegand. Using Intelligent Agents to Manage Business Processes. In *Proceedings of The First International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 345-360, The Practical Application Company Ltd, 1996.
49. N. R. Jennings, E. H. Mamdani, J. M. Corera, I. Laresgoiti, F. Perriolat, P. Skarek, and L. Z. Varga. Using ARCHON to develop real-word DAI applications, Part 1. *IEEE Expert*, 11(6):64-70, 1996.
50. M. Koster. The Web Robots Pages. http://info.webcrawler.com/mak/projects/robots/robots.html, 1996.
51. P. N. Kugler and M. T. Turvey. *Information, Natural Law, and the Self-Assembly of Rhythmic Movement*. Lawrence Erlbaum, 1987.
52. R. Levy, K. Erol, and J. J. Howell Mitchell. A Study of Infrastructure Requirements and Software Platforms for Autonomous Agents. In *Proceedings of iCSE'96*, 1996.
53. J. Maley. Managing the Flow of Intelligent Parts. *Robotics and Computer-Integrated Manufacturing*, 4(3/4):525-30, 1988.
54. Metra. *Agent Technology: UNITY_Agent: An Agent Enabler*. Metra Corporation, San Jose, CA, 1997.

55. J. Mori, H. Torikoshi, K. Nakai, K. Mori, and T. Masuda. Computer Control System for Iron and Steel Plants. *Hitachi Review*, 37(4):251-8, 1988.

56. K. Mori, H. Ihara, Y. Suzuki, K. Kawano, M. Koizumi, M. Orimo, K. Nakai, and H. Nakanishi. Autonomous Decentralized Software Structure and its Application. In *Proceedings of Fall Joint Computer Conference*, pages 1056-63, 1986.

57. D. Morley. Painting Trucks at General Motors: The Effectiveness of a Complexity-Bsed Approach. In Ernst & Young, editors, *Embracing Complexity*, pages 53-58. Ernst & Young, Boston, MA, 1996.

58. R. N. Nagel and R. Dove. 21st Century Manufacturing Enterprise Strategy. Bethlehem, PA, Agility Forum, 1991.

59. NIIIP. Mother NIIIP Homepage. http://www.niiip.org, 1996.

60. T. J. Norman, N. R. Jennings, P. Faratin, and E. H. Mamdani. Designing and implementing a multi-agent architecture for business process management. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III: ECAI'96 Workshop on Agent Theories, Architectures, and Languages*, vol. 1193, Lecture Notes in Artificial Intelligence, pages 261-275. Springer, Berlin, 1996.

61. ObjectSpace. *ObjectSpace Voyager, General Magic Odyssey, IBM Aglets: A Comparison*. ObjectSpace, Inc., Dallas, TX, http://www.objectspace.com/voyager/VoyagerAgentComparisons.PDF, 1997.

62. ObjectSpace. Voyager(tm) Core Package Technical Overview. http://www.objectspace.com/Voyager/VoyagerTechOviewOnlineVersion.PDF, ObjectSpace, Inc., Dallas, TX, 1997.

63. ODVA. Learn About DeviceNet. http://www.industry.net/c/orgunpro/odva/dev-net, 1997.

64. OMG. *The Common Object Request Broker: Architecture and Specification, Revision 2*. OMG Technical Document formal/97-02-25, http://www.omg.org/corba/corbiiop.htm, Object Management Group, 1996.

65. L. Overgaard, H. G. Petersen, and J. W. Perram. Motion Planning for an Articulated Robot: A Multi-Agent Approach. In *Proceedings of Modelling Autonomous Agent in a Multi-Agent World*, pages 171-182, Odense University, 1994.

66. Parallax. BASIC Stamp FAQs. Parallax, Inc., ftp://ftp.parallaxinc.com/pub/acrobat/stamp_faqs.pdf, 1997.

67. H. V. D. Parunak. Manufacturing Experience with the Contract Net. In M. N. Huhns, editor, *Distributed Artificial Intelligence*, pages 285-310. Pitman, London, 1987.

68. H. V. D. Parunak. *Case Grammar: A Linguistic Tool for Engineering Agent-Based Systems*. ITI Technical Memorandum, http://www.iti.org/~van/casegram.ps, Industrial Technology Institute, Ann Arbor, 1995.

69. H. V. D. Parunak. Visualizing Agent Conversations: Using Enhanced Dooley Graphs for Agent Design and Analysis. In *Proceedings of ICMAS'96*, pages 275-282, 1996.

70. H. V. D. Parunak. Workshop Report: Implementing Manufacturing Agents. *In Proceedings of PAAM'96*, 1996.

71. H. V. D. Parunak. 'Go to the Ant': Engineering Principles from Natural Agent Systems. *Annals of Operations Research*, 75:69-101, 1997. Available at http://www.iti.org/~van/gotoant.ps.

72. H. V. D. Parunak. RAPPID Project Index Page. http://www.iti.org/cec/rappid/, 1997.

73. H. V. D. Parunak, A. D. Baker, and S. J. Clark. The AARIA Agent Architecture: An Example of Requirements-Driven Agent-Based System Design. In *Proceedings of First International Conference on Autonomous Agents (ICAA-97)*, 1997.

74. H. V. D. Parunak, J. Kindrick, and B. Irish. Material Handling: A Conservative Domain for Neural Connectivity and Propagation. In *Proceedings of Sixth National Conference on Artificial Intelligence*, pages 307-311, American Association for Artificial Intelligence, 1987.

75. H. V. D. Parunak, J. Sauter, and S. J. Clark. Specification and Design of Industrial Synthetic Ecosystems. In *Proceedings of Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, Springer, 1997.

76. H. V. D. Parunak, A. Ward, M. Fleischer, and J. Sauter. A Marketplace of Design Agents for Distributed Concurrent Set-Based Design. In *Proceedings of ISPE/CE97: Fourth ISPE International Conference on Concurrent Engineering: Research and Applications*, 1997.

77. H. V. D. Parunak, A. Ward, M. Fleischer, J. Sauter, and T.-C. Chang. Distributed Component-Centered Design as Agent-Based Distributed Constraint Optimization. In *Proceedings of AAAI Workshop on Constraints and Agents*, pages 93-99, American Association for Artificial Intelligence, 1997.

78. H. V. D. Parunak, A. Ward, and J. Sauter. A Systematic Market Approach To Distributed Constraint Problems. In *Proceedings of International Conference on Multi-Agent Systems*, pages (submitted), AAAI, 1998.

79. F. Perriolat, P. Skarek, L. Z. Varga, and N. R. Jennings. Using ARCHON, Part 3: Particle Accelerator Control. *IEEE Expert*, 11(6):80-86, 1996.

80. J.-F. Perrot. Preface. In J. Ferber, editor, *Les Systèmes Multi-Agents: Vers une intelligence collective*, pages xiii-xiv. InterEditions, Paris, 1995.

81. R. Roberts. *Zone Logic: A Unique Method of Practical Artificial Intelligence*. Radnor, PA, Compute! Books, 1989.

82. G. C. Roman. A Taxonomy of Current Issues in Requirements Engineering. *IEEE Computer*, (April):14-22, 1985.

83. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, Prentice Hall, 1991.

84. M. J. Shaw and A. B. Whinston. Task Bidding and Distributed Planning in Flexible Manufacturing. In *Proceedings of IEEE Int. Conf. on AI Applications*, pages 184-89, 1985.

85. Sun. JavaSoft Home Page. http://java.sun.com, 1997.

86. T. Takashina and S. Watanabe. The Locality of Information Gathering in Multiagent Systems. In *Proceedings of Second International Conference on Multi-Agent Systems (ICMAS-96)*, pages 461, 1996.

87. K. J. Tilley and D. J. Williams. Modelling of Communications and Control in an Auction-based Manufacturing Control System. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 962-967, IEEE, 1992.

88. TPAC. The Practical Application Company. http://www.demon.co.uk/ar/TPAC/index.html, 1997.

89. A. Ward, J. K. Liker, J. J. Cristiano, and D. K. S. II. The Second Toyota Paradox: How Delaying Decisions Can Make Better Cars Faster. *Sloan Management Review*, (Spring):43-61, 1995.

90. J. E. White. *Telescript Technology: The Foundation for the Electronic Marketplace*.

1994.

91. J. E. White. Telescript: Transportable Agent Systems. http://www.genmagic.com/Telescript/, 1996.

92. J. E. White, C. S. Helgeson, and D. A. Steedman. *System and method for distributed computation based upon the movement, execution, and interaction of processes in a network*. General Magic, Inc., U.S.A., 1997.

93. R. Zanconato. An inter-agent communication model for real-time Distributed AI Applications. In *Proceedings of PAAM-96: First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 755-771, The Practical Application Company, 1996.

94. Z-World. About Z-World. http://www.zworld.com/about.html, 1997