

Practical and Industrial Applications of Agent-Based Systems

H. Van Dyke PARUNAK

Industrial Technology Institute

1. Introduction

DAI techniques have been applied to a wide range of entities laying claim to the title of “agent.” There are almost as many definitions of “agent” as there are researchers. For some, a software module is an agent only if it can travel over a network. For others, agenthood means that the module takes action on behalf of a human user. Still other researchers insist on a certain minimal level of intelligence, or the use of a specified inter-agent language, or the ability to manipulate explicit models of beliefs, desires, and intentions. The lowest common denominator of almost all definitions of “agent,” and the definition used in this chapter, is a “proactive object.” That is, a software entity is an agent if it has the data and code encapsulation of a software object, its own thread of control (making it an active object), and the ability to execute autonomously without being invoked externally (thus proactive rather than reactive). More formally [77], as illustrated in Figure 1, an agent is a bounded process, a subset of whose state variables are coupled across the boundary to the state variables of the environment. As shown in the figure, it is useful to conceive of agents as interacting with one another only by way of their shared environment (even when this environment is simply a telecommunications network).

While this chapter focuses on industrial applications, it is both broader and narrower than this definition might suggest. It is broader because I have included selected development projects that are not yet industrial strength, but embody industrially important concepts or are being conducted in a way I judge likely to lead to deployable technology. It is narrower because I have emphasized agent applications in manufacturing and physical control over other fielded industrial applications such as Web-based information-gathering agents¹ or business planning agents². I am better acquainted with the domain of manufacturing and control, the problems of interfacing agents

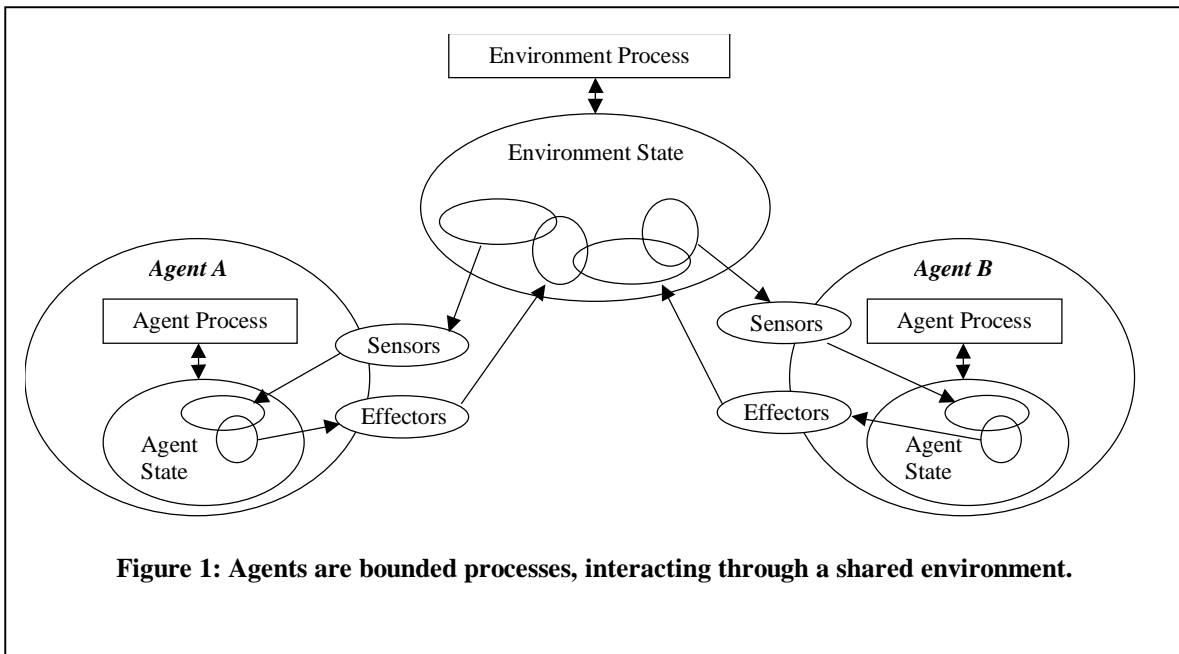


Figure 1: Agents are bounded processes, interacting through a shared environment.

¹ A convenient sampling is described in Appendix G of [13], and available on-line at [53].

² An outstanding example is the ADEPT project [48, 50, 51, 66].

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 2 develops a taxonomy of agent architectures to provide a vocabulary for describing individual industrial applications. Section 3 describes the main industrial motivations for choosing an agent architecture for a particular problem. Section 4 describes the concept of a system life cycle, and uses this concept to organize case studies of industrial agent-based systems. Section 5 returns to the taxonomy of agent architectures and identifies the alternatives most commonly selected for industrial agent-based systems.³

2. What kinds of agent architectures are there?

This section provides a brief taxonomy of the kinds of agent technology in industrial use, to provide vocabulary for discussing later cases. In comparing different alternatives, it is useful to distinguish between characteristics of individual agents, and characteristics of the overall agent community. [77] discusses the relative advantages and disadvantages of some of the alternatives identified here, based both on theoretical considerations and on reverse engineering of natural agent systems.

2.1 Characteristics of Individual Agents

In describing individual agents, it is important to understand how agents are mapped onto the problem domain, how agents model their environment, and how they are organized internally.

2.1.1 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.” Carried over to agent-based systems, these instincts lead to individual agents responsible for such functions as Logistics, Transportation Management, Order Acquisition, Resource Management, Scheduling, and Dispatching [6, 33].

The functional approach is particularly well suited to centralized systems, but unprecedented in naturally occurring systems, which divide agents on the basis of distinct entities in the physical world rather than functional abstractions in the mind of the designer. Our experience with agent-based prototypes supports this principle. Each functional agent needs detailed knowledge of many of the physical entities being managed, and so when the physical system changes, the functional agent needs to change as well. However, it is often possible to endow physically defined agents with generic behaviors from which the required functionality will emerge, for widely varying overall populations of agents.

There are two important exceptions to this principle: legacy systems and watchdogs.

Most industrial agent applications are additions to existing systems. They need to interface with *legacy systems*, many of which are functionally oriented. For example, a shop-floor control system needs to interface with a factory-wide MRP system that is doing classical scheduling. A reasonable way to connect the legacy system to the new system is to encapsulate it as an agent. Even though the MRP system may be functionally defined, as a legacy program it is a well-defined “thing” in the application domain and so deserves agenthood. Its presence in the system will not preclude the development of techniques for emergent scheduling as long as its agent behaviors restrict it to serving as a link with the rest of the system rather than drawing on its centralized scheduling behavior.

³ In the interest of concreteness, this chapter contains a large number of case studies of actual systems. This selection is meant to be exemplary, not exhaustive, and is largely biased by the accidents of discovery and the problems to which our group has sought to apply agent technology. My description of each case is based on my own reading of the case’s main publications. In some cases but not others I have had the benefit of personal discussion or email correspondence with principals in the system being discussed, but I am solely responsible for any misunderstandings or errors in description.

Some system states that local agents cannot perceive may need to be monitored to ensure overall system safety or performance. A *watchdog agent* that simply monitors the behavior of a population of physical agents is not nearly as restrictive on future reconfiguration as one that does centralized planning and action. It is best not to rely on watchdogs at all, but if they are used, they should sense conditions and raise signals but not plan or take action.

2.1.2 How does each agent model the world?

[40] argues persuasively that any agent that functions in a changing world must model that world internally. However, agents differ in the sophistication of the knowledge representation and reasoning they use for this task. Sometimes an agent models aspects of its world explicitly, so that it can reason about the model. In other agents, these models are hard-wired into the structure of the agent, and often distributed throughout the agent's architecture [27]. In addition to the implicit-explicit distinction, we distinguish two other dimensions in the models that agents use: their domain (environment vs. other agents), and their time (present vs. future).

The world in which an agent lives includes both other agents and diffuse, unbounded processes that are not usually considered agents. For example, the business activity in a retail outlet depends both on the financial state and resource requirements of the customer agents, and on the weather (an aspect of the non-agent environment). In manufacturing, an agent accesses the (non-agent) environment via sensors and actuators, and interacts with other agents via communications networks. An agent may maintain explicit models of neither of these aspects of the world, of only one or the other, or of both.

From the perspective of time, an agent may model the world as it is now, or as it wishes the world to be. A widely used architecture for agents requires explicit modeling of beliefs, desires, and intentions (so-called "BDI agents") [84]. In terms of our taxonomy, beliefs are models of the world as it now is, while desires and intentions concern a future state of the world. An external observer can attribute beliefs, desires, and intentions to any agent, based on its behavior, but a BDI agent is one that maintains explicit models of the world.

2.1.3 How are agents structured internally?

Agents differ from one another in their internal structure along four dimensions.

Similarity: Agents that do not communicate directly with one another, but simply interact through their effects on the world, may differ widely from one another in their internal architecture. If they communicate with one another, they must speak the same language. Sometimes this commonality extends to the entire agent, so that agents all run the same code and differ only in state parameters. More commonly, they share identical modular heads (including at least communication interfaces, and perhaps other capabilities such as modeling of associates, as in the GRATE* architecture [52]), but the code in their bodies is architecturally distinct from the head and may differ from one agent to the next. Thus we can distinguish among dissimilar, identical, and body-head agents.

Modularity: The body-head model permits reuse of communication and modeling code among agents that are otherwise dissimilar. Some agent environments, such as BRIC [27] or Cybele [56], facilitate the assembly of agents from pre-existing modular components. This type of structure lies at the heart of the subsumption architecture [9].

Memory: Agents may or may not retain a trace of changes in their state based on their experiences.

Mutability: In some systems an agent's code can change during the agent's lifetime. "Code" means a data structure that is executed through time. A simple linear sequence of instructions does not count; there must be some branching or decision-making. The modification may either be imposed on the agent from outside or initiated internally.

2.2 Characteristics of the Agent Community

Different agent systems differ at the community level in the number of agents, the communication channels and protocols that they use, how the agents are configured relative to one another, and how they coordinate their activities as the system runs.

2.2.1 How many agents are there?

Systems differ on how large the population of agents is and the ways in which it can change over time. Both the number of different agent species and the total number of individual agents are important characteristics of a given system. The agent population can change while the system is running in four different ways, and systems differ in which of these they support. Agents may be created, they may be destroyed, multiple agents may fuse into a single agent, and a single agent may divide into multiple agents. Agent division is distinct from agent creation in that created agents are initialized to some standard state, while agents produced through division retain some aspect of the parent's state, as a process does under a UNIX® fork.

2.2.2 What communication channels do agents use?

The channels through which information moves from one agent to another can be distinguished with respect to medium, addressing, persistence, and locality.

Medium: Agents may communicate only through changes in their shared physical environment, or (more commonly) they may exchange digital messages over a communications network. Even in this latter case, it is important to account for information that flows through the non-digital world, since this information is part of the overall communication protocol.

Addressing: Messages may be broadcast to the entire agent community, or restricted to some subset of the population. Two important forms of restricted addressing are direct addressing (in which one agent specifies the specific recipients of a message) and subject-based addressing (in which messages are labeled by content rather than by recipient, and are forwarded to all agents subscribing to the specified content area).

Persistence: Messages may persist for a period after they are sent (as in a blackboard system), or may exist only transiently, as they travel over the network.

Locality: Some agents can move in communications space so as to change their communicative proximity to one another. The key issue in this category is communicative proximity. Mobile robots that communicate with one another over a broadcast radio link do not "travel" in this sense, since their communications relationships are not affected by their movement. But an agent that represents a part in a factory, one with an escort processor that communicates only with readers that it approaches physically, is really a kind of "smart message" and does fall under this category.

2.2.3 What communications protocols do agents use?

A communications protocol determines how conversations among agents are structured.

Directive: Agents give orders to one another.

Voting: Agents express themselves by a scalar quantity, and their behavior is determined by some measure of aggregation (such as a sum, product, or average) over the scalars of different agents.

Negotiation: As in voting, negotiating agents exchange a series of messages before a decision is made. The canonical example is the contract net [20]. Voting and negotiation differ in three ways. First, while both a bid and a vote are intended to influence the behavior of the recipient, the bid has the additional purpose of acquiring further assignments for the sender. Second, the exchanges in voting tend to be shorter than those in negotiation, since the protocol must include additional steps for awarding the contract and managing the execution of the task. Third, while the simplest bids (like votes) are just scalars, bids can become much more complicated, including extensive symbolic information representing the timing and specifications of the task to be performed.

Speech Acts: Negotiation protocols determine a fixed set of options that a conversation can follow, and this structure is fixed when the agents are implemented. More recently, speech act theory has been used to build general grammars out of which arbitrary protocols can be constructed, and agents that understand these grammars can evolve new protocols for their conversations as they operate.

2.2.4 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 can be established in two ways. It may be set in advance by the system implementer, and thus remain unchanged as the system operates. Or the agents may be able to discover new relationships and configure themselves as the system operates.

2.2.5 How do agents coordinate their actions?

Agents are autonomous in that they do not have to be invoked in order to execute. However, they are not anarchical. Their actions are coordinated with one another, through one or another of a number of mechanisms. Coordination is sometimes refined into more specific categories of cooperation and competition. This distinction is sometimes hard to make. For example, competitors in a market economy actually strengthen one another, and thus at a higher level may be considered cooperative. It is more useful to distinguish hierarchical coordination (in which commands flow down from higher levels and status information flows back up) and egalitarian or heterarchical coordination [38] (in which coordination emerges from the dynamics of agent interaction).

In systems with hierarchical coordination, it is useful to observe the depth of the hierarchy, and the fan-out at each level.

Two mechanisms have been used to induce coordination in systems in which agents are not wired into positions of authority over one another: dissipative fields and constraint propagation. In a dissipative field, the environment supports a scalar flow field that agents perceive. They orient themselves to the slope of this field and reinforce it by their actions. Examples include currency-based markets [16] and pheromones in insect societies. In constraint propagation, agents pass symbolic structures among themselves and orient themselves based on the contents of these structures. [82] describes a system that combines these dynamics.

3. Why use DAI in industry?

Agents are not a panacea for industrial software. Like any other technology, they have certain capabilities, and are best used for problems whose characteristics require those capabilities. Five such characteristics are particularly salient: agents are best suited for applications that are modular, decentralized, changeable, ill-structured, and complex.⁴ In some cases, a problem may naturally exhibit or lack these characteristics, but the modern vogue for business process reengineering suggests that 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.

3.1 Modular

Agents are pro-active objects, and share the benefits of modularity that have led to the widespread adoption of object technology. They are best suited to applications that fall into natural modules. To see this more clearly, recall that 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. Multi-agent systems are thus a good approach to industrial processes that can be partitioned into a set of such entities.

The state-based view of the distinction between an agent and its environment helps us understand why 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.

⁴ These are an extension of the categories described by [45].

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.

3.2 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.

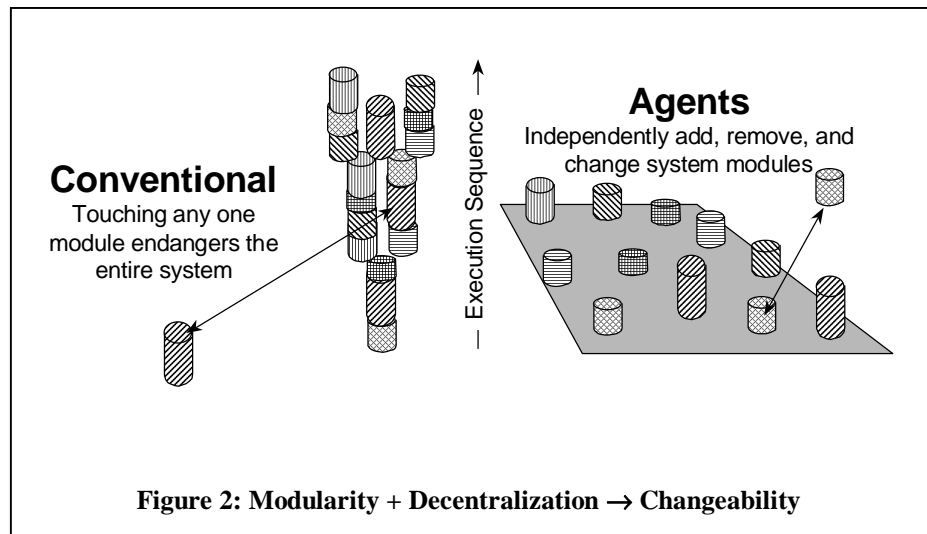
Many industrial processes can be organized in either a centralized or a decentralized fashion. Centralized organizations go back to the imperial governments of ancient Egypt, Assyria, China, and Babylon, with their focus on a central demigod and an elaborate bureaucracy to manage the flow of control 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].

This approach is not the only alternative. 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). 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 [63]. Agent-based architectures are an ideal fit to such an organizational strategy. In fact, a European observer suggests that one of the forces leading to the growing popularity of multi-agent systems is “the rise of the American style of liberalism and individualism” [83].

3.3 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 2 suggests,⁵ 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.



⁵ 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, and one of the most effective means known to determine what product features customer 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 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.

3.4 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.

Such an application is a natural one for agents. 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.

3.5 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 consider coding a separate subroutine for each possible routing, but this approach quickly becomes prohibitive as the shop grows.

As is often the case, the complexity in this example is combinatorial in nature, resulting from the fact that 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.

Just as well-structured systems can become ill-structured when viewed over their entire life span, so a system that appears to require only a few behaviors can become more complex as it is modified in response to changing user requirements. 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.

4. What characterizes the industrial world?

Industrial applications differ from research ones in that they are a means to an end, not an end in themselves. An important manifestation of this perspective is that 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.

There are two ways in which the life cycle perspective helps us understand the current state of practice in industrial multi-agent 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?

4.1 Overview of the Industrial Life-Cycle

Any activity or system in a business enterprise follows a life cycle, beginning with identifying the needs that the activity or system is intended to satisfy and ending with the completion of the activity or retirement of the system. To keep the exposition concise, we will use the term “project” to represent a specific system or activity, and illustrate the successive phases from two domains: a physical system (a new automobile), and a software system (a new factory scheduling system). The physical system bifurcates at the design phase into two systems, one concerned with the product itself, the other concerned with the system 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 here is not on *what* the project will do, much less on *how* it will do it, but on *why* an effort is needed in the first place.

- Physical: Market analysis reveals that we are losing sales to competitors who are offering sport utility vehicles (SUV's), 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 (WIP) 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 SUV offering. Further study shows that we are aiming for a consumer market, not an industrial one, so the SUV 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, MIS is notoriously nonresponsive to plant needs, and their data is usually wrong, so we have no confidence that a scheduling project that grows out of our existing MIS 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.

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 characteristics of our competitors' SUV offerings to determine what customers do and do not like, and to identify features we can add to differentiate our product in the SUV marketplace. The result is a list of the features and performance characteristics of the new vehicle.
- Software: A collection of shop-floor war stories enables us to identify a set of issues that can explain the poor performance, including no way to schedule preventive maintenance (PM) (leading in turn to reduced PM 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 excessive job classifications that prevent operators from filling in for 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 SUV, 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. It is instructive to note that at this point the automotive project actually becomes two projects: one to produce the vehicle, the other to produce the factory that will make the vehicle.
- Software: The manufacturing systems group, having read this chapter, 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 typically moves directly from Design to Commissioning without an intervening Implementation.

- Physical (System): Purchasing negotiates contracts for the equipment needed to construct the new SUV. 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 (System): The factory produces its first SUV.
- 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 (System): 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 (System): 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.

4.2 Where in the life cycle are agents used?

Mature real-world agent applications have been used to support two phases of the life cycle of industrial systems: design and operation. In this section, the life cycle is that of some (usually non-agent) system or product that agents are helping to realize. For example, in automotive, agents might help design a new vehicle, operate the plant that manufactures it, and maintain it when it fails.

The case studies in this section are of agent-based systems. Each one discusses the problem the system is meant to address, gives a narrative summary of the agent approach, then describes the individual agents and the agent community along the characteristics developed in the taxonomy above. The applications vary in maturity, and are described here using the following categories:

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

4.2.1 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. Each of these dimensions can provide the modularity needed for an agent-based approach. As suppliers take increasing responsibility for the detailed design of the subsystems they supply, the problem becomes increasingly decentralized as well. Designers begin work with a picture of what is required but not details on how it is to be produced. Often the "what" that is desired turns out to be prohibitively 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 value there is in reconfigurable agents to represent the various components, designers, and tools. The increased complexity embodied in modern products also favors the combinatorial benefits of an agent-based design system.

In addition to the systems outlined briefly here, Chapter 14, "Enabling Technologies for Concurrent Engineering and Collaborative Enterprises: Focus on Web, Java and CORBA" by Jagannathan and Kankanahalli describes some of the technologies that make distributed design systems possible.

4.2.1.1 Case: ACDS

The problem of designing a product that is manufactured from discrete components can be divided into two parts: designing the lowest level of components (such as the sheet metal panels for a car, or an integrated chip for a computer), and determining how individual components relate to one another to build up successively larger subsystems. Designing and fabricating completely new components is complicated and expensive, and in many cases designers can select most or all of the needed components from catalogs. This restricted problem, which bypasses design of low-level components, is known as “configuration design.” The challenging aspect of configuration design is representing and managing the constraints among component classes, individual components, and the requirements of the overall system. For products with many components, the number of potential combinations of parts can explode. ACDS (the Automated Configuration Design Service) [19] identifies a basic mapping of agents onto the configuration design domain that shows promise in reasoning through this combinatorial thicket.

Agent Mapping: ACDS uses four species of agents. A *system agent* represents the specifications of the overall system. Each *catalog agent* represents a set of parts of a given category, such as motors, cables, or pulleys. Catalog agents record the attributes for the different parts of each type, and *constraint agents* represent the constraints among components based on those attributes. A single *bid agent* monitors the evolution of the feasible set of solutions as the constraints are evaluated, tightening underconstrained spaces and backtracking if the space becomes overconstrained.

Agent Modeling: Only the bid agent does any significant modeling of other agents, as it seeks to guide the network toward convergence.

Agent Structure: Within a single species, all agents are identical, and do not mutate as the system runs. Agents do maintain memory of the result of previous interactions in the form of local state.

Population: An instantiation of ACDS includes a single system agent, a single bid agent, as many catalog agents as there are distinct component types in the overall product (on the order of a dozen in demonstration problems), and as many constraint agents as there are constraints (again, on the order of a dozen).

Communication Channels: Agents communicate electronically, using point-to-point transient messages over a network that is defined when the problem is configured. Agents do not migrate as the system operates.

Communication Protocols: Messages among agents are directive, conveying local state information so that other agents can take it into account in their reasoning.

Configuration: The structure of the agent network is fixed when the problem is configured.

Coordination: Coordination is achieved in an egalitarian manner, through propagation of constraints.

Maturity: ACDS has been prototyped on the design of the VuMan2 wearable computer developed at CMU.

4.2.1.2 Case: PACT

Design of a complex system requires the exchange of information among various automated design tools. The representation of the problem that each tool uses may be different from those required by the other tools, containing different kinds of information expressed in incompatible ways. PACT (the Palo Alto Collaborative Testbed) [18] demonstrates how turning individual tools into agents can enable them to exchange information even though they were originally designed as stand-alone applications.

Agent Mapping: PACT assigns a tool agent to each design tool, and a facilitator agent to each representation. Each facilitator agent is responsible for translating between the representation for which it is responsible and a common knowledge interchange language.

Agent Modeling: Each tool models the current state of its own domain and what it is trying to achieve, but does not explicitly model the other agents in the system.

Agent Structure: Tool agents are ideosyncratic. Facilitator agents share a common architecture for their communication module (called a “connection associate”), but have an “agent manager” module customized to the

requirements of tool agents that use its representation. Agents do remember state but do not change their code during operation.

Population: The PACT demonstration includes six tool agents, each with its own facilitator. The six tool agents represent, respectively, digital circuitry, control software, power systems, physical plant, sensors, and a parts catalog.

Communication Channels: Agents communicate electronically, using both point-to-point and broadcast messages. Messages do not persist after being sent, and agents do not migrate during system operation.

Communication Protocols: Facilitator agents send control directives to one another directly, and translate and pass on queries for information and responses on behalf of the tool agents that they support, in a primitive speech act protocol.

Configuration: Because of the need for translation to and from the common knowledge exchange language, each tool agent is hard-wired to its facilitator, and the facilitators form a message ring. Most messages among design tools are broadcast queries and the point-to-point responses that they produce, so that the logical configuration of who is talking to whom changes dynamically.

Coordination: Coordination is egalitarian, by propagation of constraints.

Maturity: PACT has been prototyped in the design of an electromechanical manipulator.

4.2.1.3 Case: RAPPID

A designer seeks to embed a set of *functions* (e.g., optical, electromechanical, control) in an artifact with specified *characteristics* (e.g., weight, color, complexity, materials, power consumption, physical size). The functional view drives most designs, since it distinguishes the disciplines in which engineers are trained and in support of which design tools are available. 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? Others face design off against other manufacturing functions: How should we 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 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 QFD (Quality Functional Deployment) 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 (Responsible Agents for Product-Process Integrated Development) [81] uses a marketplace to set prices on each characteristic of a design. Agents representing each component buy and sell units of these 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 may 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.

Agent Mapping: The two primary classes of agents in RAPPID are the designers of the individual components and the various characteristics in which the designers trade.

Agent 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 a parameterized expression of their interest, and formulate recommendations for future action from these models.

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.

Population: A realistic application of RAPPID will have one or two dozen Component agents and on the order of a hundred Characteristic agents. Agents are not created, destroyed, divided, or fused during operation.

Communication Channels: Agents communicate digitally, and currently use point addressing. Messages do not persist outside of agents, and agents do not move over the network.

Communication Protocols: 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 Components and Characteristics is defined when the system is initialized, but Components 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.

4.2.2 Operation

Most of the operational systems described here fall in the broad category of control systems. That is, the community of agents monitors the system's trajectory through state space and adjusts operating parameters to adjust that trajectory to satisfy some overall criterion. The fundamental challenge in applying agents to such systems 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 abundance of examples in this section shows 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 control program. When the system is geographically distributed, the decentralization of the agent approach reduces data communication bottlenecks and permits local parts of the enterprise to continue operation during temporary lapses in connectivity. As competitiveness comes to depend more and more on being able to adjust a system's operation frequently to track customer requirements, the changeability of agent systems becomes more valuable. 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.

Operational systems are characterized by the need for real-time response. In addition to the systems outlined here, see Chapter 11, "Distributed Models for Real Time Decision Support," Jose Cuenca.

4.2.2.1 Case: AARIA

In discrete manufacturing, Parts move through a network of Unit Processes (UP's) and Buffers. Each UP acquires one or more input Parts from Buffers of the needed types, and engages certain Resources to produce one or more output Parts into appropriate Buffers. AARIA (Autonomous Agents for Rock Island Arsenal) [4, 78] is an agent-based system to schedule and control this sequence of operations. In AARIA, we use linguistic case theory [74] over a set of declarative sentences describing the domain to identify candidate agents, then refine this population using the requirements. Previous research on agent-based factory control and scheduling differs widely on what is represented as an agent: levels in a hierarchical decomposition of the factory [12, 73, 90], Resources [3, 39, 79, 88], or Parts [21, 57]. In AARIA, separate full-fledged agents represent parts, resources and unit processes with substantially equal intelligence and responsibility in each type of agent.

Agent Mapping: AARIA has six species of agents: resources (such as machines, tooling, and operators), part types, unit processes, management, parts, and engagements between a unit process and its resources.

Agent Modeling: A resource agent embodies any models it needs to control its associated resource, and models its commitments over time to various unit processes. Unit processes model the resource types and input parts that they require. Part type agents model the unit processes that can supply or consume their part types, and the supply and demand of that part type over time.

Agent Structure: Agents of each species have the same internal structure, with a common head module for all agents, and provision for reuse of activities across agents. Agent code does not change as the system runs, but Resource Agents and Part Type Agents maintain running aggregate information of past activities to guide future decisions.

Population: The community for an operating shop will include hundreds or even thousands of individuals. The population of resource, unit process, management, and part type agents is fixed by the implementer (though new instances can be added as the system runs); the system itself generates new part and engagement agents as they are needed.

Communication Channels: Agent communication is mainly digital, using subject-based addressing. Messages do not persist unless they are maintained by an agent. Engagement Agents and Part Agents move back and forth among the other agents with which they converse.

Communication Protocols: Agent interactions are based on fixed negotiation protocols.

Configuration: The architecture defines which classes of agents can deal with one another, but the actual individual agents that interact identify one another dynamically as the system operates.

Coordination: AARIA uses egalitarian coordination, and combines constraint inferencing (e.g., a Unit Process reasoning about when it needs its input parts) with a market economy that provides a dissipative field.

Maturity: AARIA has been demonstrated against an emulation of a manufacturing job shop.

4.2.2.2 Case: ADS

Architecturally, one of the great promises of agent-based systems is the ability to add, modify, and remove agents while the system is running. This capability is particularly important in an application such as a steel mill or a transportation network in which bringing the system down for maintenance is extremely costly. Hitachi's Autonomous Distributed System (ADS) [44, 59, 60] is motivated largely by the need to modify control elements while the system is running, and has been in operation since the mid-1980's in several sheet steel processing lines at Kawasaki Steel's Chiba works and in the operator control system of the Shinkansen (the Bullet Train). In the steel application, the system is down less than 2.6 minutes per year.

Agent Mapping: Bounded processes appear at two levels in ADS. The highest level, the "atom," corresponds to a physical computer that interfaces to the communication network. Each atom contains an arbitrary number of application modules, and five service modules to support the installation, testing, and operation of the application modules. The application modules are not strictly autonomous, since they can only run when they are invoked by one of the service modules (the execution module), but that invocation depends only on the availability of the necessary data, so any application module together with the services provided by the execution module constitutes an agent. In the Kawasaki Steel application, application modules fall into three broad subsystems: on-line information management (such as production scheduling, process data logging, and human interface), software development, and real-time control. Real-time control modules are defined geographically, according to local regions of the plant, and include tracking modules that identify the presence of product and control modules that control the processing equipment.

Agent Modeling: Application modules register their interests with one of the service modules (the Data Field module) in their atom. The Data Field module uses the resulting information to monitor the network for traffic of interest to them. The architecture does not provide any specific support for one application module to model another, although an application module could maintain such a model on its own. In the Kawasaki environment, the inclusion of software development modules in the system implies that these modules at least monitor the release number and revision status of the other currently deployed modules.

Agent Structure: The five service modules are identical in all atoms; application modules vary depending on problem requirements. At the level of the atom, the architecture is highly modular, and application modules can be replaced while the system is running. Individual application modules can retain state, although in the steel application they tend to rely on the on-line information modules.

Population: In addition to as many types of application module as are needed by the problem domain, there are five species of service modules. The Data Field Management module retrieves inter-atom messages that are of interest to modules within its atom, and publishes their messages on the network. The Execution Management module determines when an application module has the inputs it needs to operate, and activates it. The Data Consistency Management module referees messages from replicated application modules and synchronizes asynchronously received messages that are needed by a single application module. The Construction Management module receives new code for an application module and installs it, and the Built-In Test module tests a newly installed module in parallel with the previous version. One steel application has 48 atoms. The number of application modules is not published, but the system manages about 730 points of input and output, which is a rough upper limit on the number of application modules. The train application integrates 15 terminal control computers at the atom level, supporting 69 operator applications. Application modules can be added and removed as the system operates.

Communication Channels: Between atoms, messages travel on a fiber-optic ring, and are all labeled by content, not destination. Within an atom, messages are addressed from one module to another. Messages do not persist. The ability to install and remove application modules while the system is running may be viewed as a form of agent migration, but the published examples do not use this capability to move an executing agent around the plant.

Communication Protocols: Application modules react to directive messages. The Data Consistency Management module uses voting schemes to resolve contradictory messages.

Configuration: The physical configuration of atoms is fixed, but the use of subject-based addressing means that the logical interactions of application modules can change dynamically.

Coordination: Within an atom, coordination is hierarchical. Between atoms, it is egalitarian, based on propagation of constraints.

Maturity: ADS has been supporting commercial production for over ten years, and is implemented on widely-used commercial platforms. The agents at Kawasaki Steel are written in Fortran, and those in the Shinkansen system are written in Cobol.

4.2.2.3 Case: AMROSE

Many ocean-going vessels have double hulls (two complete steel surfaces with a space between them). The assembly of such hulls requires some operations to be carried out between the hulls after both layers are in place. Maneuvering tools in the narrow space between the hulls (for example, to apply a weld bead along a seam) is a challenge for conventional tooling. AMROSE [71] controls an articulated robot arm with 19 segments that can snake its way through the inter-hull space and maneuver along a work track. Each joint in the robot is an agent that derives its positioning goal from the next agent closer to the end effector. If the joint holding the end effector can reach its goal, it does so. Otherwise it computes where it would need to be to move the end effector to the appropriate location, and passes this goal to the next joint in line, which follows the same procedure.

Agent Mapping: Each joint in the robot is an agent.

Agent Modeling: An agent models its own location, where it wants to be, and where the next joint in line wants to be.

Agent Structure: The agents are identical to one another, do not change over time, and maintain no memory of previous states.

Population: AMROSE has been applied successfully to robots with 19 joints. The agent population is fixed when the robot is configured.

Communication Channels: All messages are digital and addressed point to point. Messages do not persist, and agents do not move.

Communication Protocols: AMROSE messages are directive (“move me to location X”).

Configuration: The implementor fixes the configuration by defining number and connectivity of the robot joints.

Coordination: AMROSE uses egalitarian constraint coordination.

Maturity: AMROSE is used in production at Odense Shipyard.

4.2.2.4 Case: ARCHON

The initial motivation for the ESPRIT ARCHON project [47, 52, 95], like that for PACT in design, is the integration of pre-existing computer tools that were not originally intended to interoperate. ARCHON’s original domain is the management of a distribution grid for electrical power in northern Spain, based on inputs from four pre-existing expert systems. The generalized architecture, GRATE*, has subsequently been applied to the integration of two expert systems for diagnosing problems in a particle accelerator.

Agent Mapping: GRATE* assigns one agent to each expert system to be integrated.

Agent Modeling: The individual expert systems maintain as-is and to-be models of their environment. GRATE* defines a wrapper for each expert system that maintains models of the capability of other agents in the system.

Agent Structure: The individual expert systems differ from one another, but GRATE* defines a wrapper, or head, with a common architecture. This head is modular, composed of a communication manager for network interface, a cooperation module to establish and maintain cooperative activity with other agents, a situation assessment module to determine whether the local agent needs to cooperate with other agents, a control module to interface with the existing code, explicit models of the capabilities of the local agent and of its acquaintances, and an information store for operational data. Individual agent code does not change over time.

Population: The power distribution system has four agents, each of a different species: a control system interface, a black-out area identifier, an alarm analyzer, and a service restoration planner. The particle accelerator application has two agents, one to diagnose problems at the level of the particle beam, and one to diagnose problems in the control software.

Communication Channels: Messages in ARCHON/GRATE* travel digitally and are addressed to specific recipients, based on the models each agent maintains of the capabilities of its colleagues. Messages do not persist, and agents do not travel.

Communication Protocols: Most interchanges involve requests and responses, following a primitive speech act protocol.

Configuration: The configuration of agents is defined when the system is configured.

Coordination: Coordination is egalitarian, based on the propagation of the needs of each agent to others able to satisfy them.

Maturity: The GRATE* architecture has been applied to a number of prototype problems, and is in operation in an electric utility control room in northern Spain.

4.2.2.5 Case: Daewoo Scheduling System

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 physical manipulations involved in setting up and running such a press are 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 the storage area and aligned precisely with the press before parts can be made. Dies can wear with use, requiring periodic refurbishing. In spite of these setup constraints, the type of part being produced must change frequently to provide the vehicle assembly operation with the right components for the vehicles currently being manufactured.

The press shop at Daewoo Motors' integrated automobile production facility in Korea 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 producing more than 500 different parts. To make these parts, more than 2000 dies are needed. Efficient operation of this shop requires scheduling the presses, sheet metal stock, and dies in a way that is responsive to the requirements of vehicle assembly and that minimizes setup time. Common industry practice schedules such shops manually, but Metra has recently deployed an agent-based scheduling system for this shop [15].

Agent Mapping: Task agents represent individual work orders, and resource agents 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: Each domain agent has a friend module in which it caches information about its colleagues that it obtains in the course of interaction. For example, a resource agent caches some 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. 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, over the network, 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. Messages do not persist, and agents do not travel.

Communication Protocols: This application 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.

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

4.2.2.6 Case: GM Paint Shop [24]

Control systems for industrial equipment are often programmed in a rule-based language known as "ladder logic," derived from wiring diagrams for the physical relays that controlled automation a hundred years ago. In such physical relay systems, each statement (or "rung") of ladder logic described a possible circuit from an energized wire to ground, specifying a set of relay contacts in series with a set of relay coils. The electrician would wire the relays so that when the specified contacts close, the specified coils are energized. Though physical relays have been mostly replaced with digital controllers, this same circuit metaphor is still widely used in describing control actions. The contacts are replaced with an input set of sensors or internal conditions, and the coils with an output set of actuators that will be energized or conditions that will be set if the input set is satisfied. Because of the sequential execution of a von Neumann machine, the rungs of logic are usually executed in fixed sequential order, beginning again with the first after the last has executed.

The typical approach to programming an industrial system seeks to make each rung functionally complete and independent. For example, the control of paint flow to a paint robot might traditionally be conditioned not only on the presence of a part in the paint zone, but also on the state of the air blowers, the presence of high voltage between the gun and the part (to attract the paint to the part), the movement of the conveyor, an indication that the system is in automatic mode, and the lack of an emergency stop condition. In the agent-based approach, the air

blowers, high voltage system, and conveyor are separate agents that determine their own operation and are not “supervised” by the gun control logic. The automatic mode logic directly enables the proximity switch that requests paint flow, and so need not be queried directly. Similarly, an emergency stop cuts off all power to the equipment, so there is no need for the gun controller to communicate directly with the emergency stop logic. As a result, the logic for the paint gun is simply to turn on when it receives a “part present” signal and turn off otherwise, trusting the environment to convey the information that the other subsystems are operating correctly.

The Flavors Parallel Inference Machine described in Section 4.3.3.1 is an industrial-strength parallel computer that breaks the sequential ordering of execution logic. Its ParaCell language replaces the wiring metaphor with a direct rule-based paradigm. Rules are grouped into cells, and all cells execute concurrently.

The agent-based design for paint gun control outlined above has not been implemented, but this approach was applied successfully to the control of the air supply for the paint booths at General Motors’ Fort Wayne truck assembly plant. The humidifiers, burners, and steam generators for the air supply system were controlled as separate agents, each reacting autonomously to various environmental features rather than querying one another’s control logic directly. The result is a \$2M annual savings in paint, reduction in the amount of control software by 40%, reduction in system development time to four days and system commissioning time to six days, and an overall system simple enough to be maintained by an electrician rather than requiring an engineer.

The major conceptual breakthrough in the Fort Wayne paint shop application is treating each device as responsible for its own actions and trusting the physical environment to maintain its own state information for consultation by the devices in making their decisions.

Agent Mapping: Each device or subsystem (humidifier, burner, chiller, steam generator) is an independent agent.

Agent Modeling: Agents do not model themselves, the environment, or other agents explicitly, but contain hard-coded rules connecting environmental conditions with appropriate actions.

Agent Structure: Agents are separate monolithic pieces of code.

Population: The system uses fewer than a dozen agents.

Communication Channels: Communication in the PIM is through shared memory, mapped onto physical sensors that reflect the actual state of the environment.

Communication Protocols: The paint booth agents are reactive.

Configuration: The physical arrangement of agents is determined by the process restrictions. Their logical interconnection is not specified in advance, but emerges from the environmental conditions consulted by each device in making its individual decisions.

Coordination: Coordination is heterarchical, through propagation of constraints via the environment.

Maturity: This system is in production use at General Motors’ truck assembly plant in Fort Wayne, Indiana.

4.2.2.7 Case: HP's JetSend [41]

One way to view many digital devices is as information appliances. Digital devices and scanners produce information; printers, screens, and video projectors display it for human perception; user programs edit and search it. These various differ in the kinds of information they can handle (e.g., text vs. images vs. sound vs. touch, still vs. motion, black and white vs. color, various levels of resolution). Currently, a central device (the “computer”) must contain device drivers that know about the capabilities of the other devices (the “peripherals”). For example, a user with a scanner and a printer has the basic elements of a photocopier, but needs to connect them through a computer that can mediate between the capabilities of the individual devices. JetSend is a protocol that enables digital devices to negotiate their own interactions without the need for a central computational platform.

Agent Mapping: Each JetSend device is an agent.

Agent Modeling: Each device knows the characteristics of the information that it produces or consumes.

Agent Structure: The JetSend protocol provides a common head to various different physical devices and their varied internal processing.

Population: A typical user might have on the order of a dozen or so appliances, typically connected in twos or threes at a time.

Communication Channels: Devices communicate digitally, broadcasting to any other devices on the line with them. Information does not persist outside of the devices themselves.

Communication Protocols: JetSend devices use a simple contract net protocol to determine the kind of information that they can exchange with one another.

Configuration: JetSend devices are typically connected together by the user, but a device with information to send can seek out a compatible partner from among available alternatives.

Coordination: Coordination is heterarchical, based on peer-to-peer constraint relaxation.

Maturity: JetSend is a commercially available technology.

4.2.2.8 Case: ICSM

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 (such as steel, copper, and plastic resin) into a vehicle is a network of many different firms. Such a network is called a “supply chain.” The company that offers the final product to the customer is sometimes called the “Original Equipment Manufacturer” (OEM). Its suppliers are called “first-tier companies,” and their suppliers are called “sub-tier companies.”

Supply chains bring both benefits and costs to firms that participate in them. The major benefit of a supply chain is its ability to respond quickly to market changes by changing participants. Each member can tailor its operations to its own specialty, and can team with other firms as needed to meet a particular market opportunity. The chain can produce a far wider variety of product options than can a single company, since it can harness different suppliers with expertise in different subsystems of the product. The major challenge of a supply chain is in coordinating activities across different organizations. This challenge applies to many phases of the life cycle. In design, the OEM must rely on its suppliers for design expertise in the systems that they supply. In operation of the production system, the flow of materials must be scheduled smoothly across all participants. In operation of the product, dealers or other maintenance providers must have access to technical expertise from the firms responsible for the various subsystems of the product.

The decomposition of a manufacturing enterprise into a network of firms and the decentralization of control that results make supply chains a natural domain for the use of agent technology. The Integrated Supply Chain Management Systems (ISCM) [32] is an extended research program at the University of Toronto devoted to exploring this opportunity. It focuses on the flow of material through the supply chain during the operation phase of the life cycle.

Agent Mapping: The major decomposition of ISCM is by enterprise function, and distinguishes eight agent species: seven functional agents and one information agent. The functional agent species are Order Acquisition, Logistics, Scheduling, Resource, Dispatching, Transportation, and Plant Management. The Information agent is a central communication resource that provides knowledge management, conflict resolution, and coordination support to the other agents. In applying the model to a given supply chain, one instance each of the Information agent and the Logistics, Order Acquisition, and Transportation agents serve the entire enterprise, and each plant has its own instance of another Information agent, the Plant Manager, Resource Management, Dispatching, and Scheduler.

Agent Modeling: ISCM agents model and reason explicitly about one another and their environment, and use AI techniques to plan how to move from the current state of the world to a desired state.

Agent Structure: The eight agent species are distinct from one another, but make use of the common resources of the Agent Building Shell, described later in this chapter. Agents have sophisticated memory capabilities and can be of arbitrary sophistication, but the applications that have been described do not focus on agent mutability as a major feature.

Population: ISCM supports eight species of agents. A supply chain will have four agents at the enterprise level and five at each plant, so a typical model can easily have forty or fifty agents.

Communication Channels: Agents communicate electronically, using direct addressing and subject-based addressing. Information agents provide persistent storage for shared information. Agents do not migrate through the system.

Communication Protocols: The Coordination Language (COOL) that defines coordination among agents supports arbitrarily complex protocols using speech act performatives.

Configuration: The relations among agent species within a single plant, and between these agents and the enterprise-level functional agents, are defined by the architecture, but the course of a given conversation will vary dynamically based on the state and capability of the individual agents.

Coordination: ISCM is coordinated by propagation and resolution of constraints among agents.

Maturity: ISCM has been emulated using the Toronto Virtual Enterprise (TOVE), a sophisticated model of the business enterprise.

4.2.2.9 Case: LMS

A semiconductor fab builds up successive layers of circuitry on a silicon wafer by passing the wafer repeatedly through a series of processes, such as oxidation (to lay down an insulating layer between layers of circuitry), photolithography (to record the traces of a layer), etching (to remove material not masked by photolithography), and ion implantation (to dope the silicon with appropriate materials). Because the same tools are used repeatedly in the process, several batches of wafers often find themselves contending for the same tool. The Logistics Management System (LMS) [30, 31] uses a set of agents at each tool to select the next batch to gain access to that tool. Each agent evaluates a different criterion for admitting a batch to the tool. One assesses Serviceability (how far behind or ahead of schedule a given lot is), one assesses Daily Planned Output (based on a quota per chip type), one assesses Downstream Pull (how critical a given wafer type is to keeping downstream resources busy), and one assesses Tool Charge, Characteristics, & Utilization (the degree to which the tool needs to be reconfigured to accommodate a specific wafer). Each agent votes on a scale of 0 to 1 for each batch of wafers that is waiting for the resource. A single 0 forces a batch to wait, a single 1 with no 0 selects a batch, and otherwise the batch with the highest total score is selected.

Agent Mapping: LMS uses functional agents, one for each production constraint, and a judge agent to combine the votes of the four critics.

Agent Modeling: Each agent models those aspects of the environment needed to satisfy its objective. The goal of each agent is hard coded, since the agent is dedicated to that goal.

Agent Structure: Each of the five species is distinct from the others, but all are body-head in that they share a common voting language. Agent code does not change as the system runs, but individual agents do remember past information as needed to assess future wafers.

Population: Each tool has five agents, one of each species.

Communication Channels: Agents communicate digitally, addressing themselves to the judge. The Downstream Pull agent also communicates with other tools in the fab. Messages do not persist and agents do not travel.

Communication Protocols: LMS uses a voting protocol.

Configuration: Agent configuration is fixed at design time.

Coordination: LMS coordination is egalitarian, driven by the relation of each batch of wafer to the four production constraints.

Maturity: LMS is used in commercial production at IBM's semiconductor fab in Burlington, VT.

4.2.2.10 Case: Market-Based Climate Control

The offices in a modern commercial building maintain their temperature by drawing on a stream of hot or cold fluid (air or water) produced by a central furnace or chiller. Each zone or office has its own thermostat, which compares the ambient temperature with the desired setting and opens or closes a valve to change the amount of hot or cold fluid that flows through the zone to modulate its temperature. On a very hot day, the amount of cooling called for by the thermostats may be greater than the cooling capacity of the central chiller, and on a cold day, the local zones may call for more heat than the central furnace can generate. In both cases, offices close to the central resource will maintain their desired settings much more closely than those farther down the pipeline, and individual offices may experience wide swings in temperature during the course of the day as aggregate demand changes in relation to overall capacity.

A system in operation at Xerox PARC [17, 18] breaks the direct connection between each thermostat and its control valve. Each thermostat receives a steady flow of funds, and on a periodic basis bids against other thermostats for access to the heating or cooling fluid. A thermostat offers to sell if its zone is receiving more fluid than needed (in the case of cooling, if the ambient temperature is cooler than the setpoint), and otherwise offers to buy. The amount of its bid is determined by how far the actual temperature is from the setpoint. In each period, a central marketplace aggregates the bids, computes a market closing price and volume, and directs the various valves to open or close on the basis this closing point. The system maintains better temperature stability than a traditional climate control system that had been tuned by facilities engineers over a period of two years.

A recent analysis of this system [96] suggests that its superior performance can be attributed to its use of centralized information, in comparing the temperature of each office to the overall average in making allocation decisions. However, this same analysis shows that even better performance can be achieved without this aspect of centralization, by refining the underlying market model.

Agent Mapping: Each thermostat is an agent.

Agent Modeling: Each agent models the as-is state of its environment and maintains a goal based on the thermostat setting. Agents do not model or reason about one another.

Agent Structure: Agents are identical except for their parameters. Their code does not change, and they maintain no memory of past actions.

Population: The PARC prototype controls 53 offices configured in 41 separate zones.

Communication Channels: Agents communicate through digital channels, dealing directly with the market server. Messages do not persist, and agents do not travel.

Communication Protocols: The system's single-bid market mechanism is a variety of voting.

Configuration: The configuration is fixed by the system designer.

Coordination: The system is coordinated by the dissipative flow of currency among agents of equal standing.

Maturity: This system is in production use at Xerox's Palo Alto Research Center.

4.2.2.11 Case: Paper and Steel Mill Scheduling [55, 62]

Products such as paper and sheet steel are produced with a batch process, then trimmed, packaged, and sold as piece products. For instance, paper production requires scheduling different alternatives in primary production (paper machines), trimming, finishing (cutting into sheets and packaging), and shipping (trucks vs. trains). IBM markets an agent-based system for this application, based on the A-team architecture in which functional agents generate, evaluate, improve, and prune a pool of candidate solutions. The approach provides a population of candidate solutions at any time. The longer the system operates, the more the population comes to lie on the Pareto boundary of the particular problem being addressed.

Agent Mapping: The agents are of four species:

1. Evaluation agents evaluate the tardiness, operating costs, and downtime of complete schedules.
2. Constructors devise new solutions, using local heuristics.
3. Improvers tweak existing solutions to make them better, using mathematical optimization algorithms, domain-dependent heuristics, and search.
4. Destroyers remove bad solutions from the pool, using heuristics such as how far a given schedule is from the Pareto frontier for the problem, or whether the schedule violates certain ad-hoc constraints.

Agent Modeling: Agents do not maintain explicit models of themselves or other agents. The degree to which they model the environment depends on the particular heuristics embodied in each agent.

Agent Structure: Each agent is a separately developed piece of code, embodying its own techniques.

Population: A typical application has on the order of a dozen agents.

Communication Channels: A-teams communicate using shared data.

Communication Protocols: The various functional agents react to the schedules available in the pool, and do not directly communicate with one another.

Configuration: The role of each agent is determined by the system developer.

Coordination: Coordination is heterarchical by constraint propagation through the pool of solutions.

Maturity: The system is available for sale as a commercial product from IBM. No details on its performance in actual operations are available.

4.2.2.12 Case: PID Control Elements

Current technology for industrial process control offers many examples of coordinated proactive objects that can usefully be viewed as agent-based systems. For instance, a typical chemical plant contains hundreds of PID control loops, each a separate computer that adjusts a setpoint as a function of various sensors, and the action of one such loop changes physical quantities that will affect the behavior of other loops in the system. PID stands for “proportional, integral, derivative,” and describes the three functions that the agent can apply to the stream of data from a sensor to determine the setpoint for a particular actuator.

Agent Mapping: One agent is assigned to each variable that can be independently adjusted.

Agent Modeling: PID elements do not maintain explicit models.

Agent Structure: PID elements are identical except for their parameters, and change during operation only as the result of human adjustment. They maintain a short-term memory of past sensor readings that can be used to compute the integral and derivative of the sensory stream.

Population: Hundreds or thousands of such elements are deployed in a typical installation.

Communication Channels: Communication among PID units is entirely by means of physical changes in the environment that are generated by one unit and sensed by another.

Communication Protocols: PID communication is entirely directive; a unit unilaterally exerts a change on its environment.

Configuration: The assignment of PID elements to process variables and the physical configuration of these elements in equipment is done by the designer when the system is designed.

Coordination: PID elements are not usually thought of as a coordinated community, but their changes to a shared environment constitute an egalitarian constraint-based process that could be analyzed and engineered as a coordinated activity.

Maturity: PID systems are widely deployed as commercial products.

4.2.2.13 Case: Zone Logic

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 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 much higher processing rates than discrete machines served by separate material transport systems. However, the large number of mechanisms that it contains pose a severe problem in coordination and control. A typical transfer line may be a hundred meters long and contain dozens of stations with hundreds of mechanisms and over 1500 degrees of freedom in movement. Traditional control schemes for such systems require the software engineer to keep all these mechanisms straight. 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.

The Zone Logic system [85] makes each mechanism in the transfer line an agent that expects a certain range of conditions and knows what to do in each. A malfunction takes the form of a mechanism that encounters a condition it does not expect, and in this case the mechanism identifies itself to the operator, permitting rapid repair. When the system is restarted, the ability of individual mechanisms to detect local conditions and act accordingly means that there is no need to reset the line to a known initial state. Transfer lines controlled with Zone Logic routinely are available 90% of the time.

Agent Mapping: Each mechanism (e.g., clamp, slide, transfer bar, probe) in the transfer line is a separate agent.

Agent Modeling: Each agent maintains a rule base listing the state conditions it recognizes and what action it should take in each case. There are no explicit models of other agents and no explicit goals.

Agent Structure: All agents use the same basic code and do not change as they run. The condition-action rules for each mechanism are different.

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 is used to guard against interference between mechanisms that may need access to the same physical space. Zone Logic agents are assigned to specific physical mechanisms installed at fixed locations on the line, and so do not 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.

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: Zone Logic agents coordinate their activity by propagating constraints.

Maturity: The Zone Logic system has been deployed as a commercial product in several automotive manufacturing facilities.

4.3 How does industry constrain the life cycle of an agent-based system itself?

The industrial environment poses restrictions and constraints that are not present in most research environments, and thus must be considered explicitly. Here again we follow the life cycle paradigm, but this time focusing on the development of the agent-based system itself. That is, now one needs to consider what constraints industrial use places on the design of an agent-based system, its construction, and other phases of the life cycle. 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 often do not exist to meet the challenges they explore.
2. Their educational mission is well served by involving students intimately in the inner functions of 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. Second, 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.

4.3.1 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. Agent technology permits us to apply computers to highly distributed, ill-structured problems that previously would not have been candidates for computer support. Thus we can now consider drafting requirements for problems that we would not have bothered to analyze before. Requirements engineers need to understand the benefits of agent-based systems over centralized monolithic systems, at least at a high level, to understand what kinds of requirements 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 of the issues 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. [86] offers a helpful summary. A complete design method for agent-based systems needs to take account of all these issues. This discussion illustrates the nature of requirements and specification by focusing on the functional requirements, those that concern the behavior of the system.

At a high level, desired system behavior may be of several kinds. We may want the system to maintain some set of state variables in a specified relationship with one another, thus exhibiting *homeostasis*. The system may be a *transducer* that needs to convert specified stimuli into corresponding responses. Or we may want the system to *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 we succeed in achieving it. 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. For simulation, we need to specify the quantitative link desired between pressure and temperature.
- It should be relevant to system architecture as opposed to other system variables. For example, the system behavior “Have tooling available when needed” might be better addressed by buying more tools rather than

expecting magic from agents. Better specifications for this example include: “Get high-value parts through the system first”; “Identify relative scarcity of tool types”; “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 intended to be exhaustive. They simply illustrate the kind of information that the requirements and specification process should produce.

4.3.2 Design

Design of an engineered artifact (such as an agent-based system) is a *process* that takes place within a conceptual *context*. For example, in automotive design, the conceptual context includes the understanding that the engineers must design a container to carry passengers or freight, some source of mechanical power, some way to transmit this power to the interface between the container and the ground, resources for providing passenger comfort and maintaining operations during varying environmental conditions, and so forth. The design process includes brainstorming sessions, development of technical drawings, construction of small-scale models, and engineering analyses (such as finite-element modeling or physical testing) of candidate designs.

Considerable attention has been devoted to defining the conceptual context within which agent-based systems are designed. In the agent research community, what I am calling a “conceptual context” is often called an “agent architecture.” Relatively less attention has been paid to the important question of the processes that the designers go through. Industrial users will use agents more readily if basic principles and guidelines are available in both areas.

4.3.2.1 The Conceptual Context

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. For example:

- The InteRRaP architecture [61] defines three control layers in each agent. The behavior-based layer defines reactive and routine responses of the agent to the world. The local planning layer defines how the agent constructs plans to address its own goals and tasks. The cooperative planning layer manages inter-agent planning, conflict resolution, task allocation, and other coordination.
- Newell [64] suggested the usefulness of a “knowledge-level” abstraction of an agent’s knowledge that is independent of the implementation details of how the agent actually stores or manipulates information. Jennings [46, 49] argues that when one moves from solitary agents to multi-agent systems, it is useful to abstract yet one level higher and describe a “social level” or “cooperative knowledge level” that abstracts away from the agent’s individual rationality.
- The design approach recommended for use with dMARS [37] requires the construction of three models: an agent model that details the knowledge, tasks, and plans of each agent; a goals model that distinguishes the purposes filled by each agent in the community or the services that it offers other agents, and a communication model that describes command, control, and communication among the agents.
- Engineers at Daimler-Benz [11] construct three major models in the process of designing an agent-based system. The agent model describes individual agents and their internal structure and dynamics. The

organizational model describes the relationships among agents and agent types. The cooperation model describes the dynamics of how agents communicate and cooperate with one another.

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 1 summarizes the various subcategories distinguished by one or another of these approaches.

Table 1: What Needs to be Designed?

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)

4.3.2.2 The Design Process

In our work with industrial clients, we have been developing an iterative refinement approach to designing agent-based systems [80]. The four stages outlined in Table 2 lead from a rough initial sketch of the community and its interactions to the point that software engineers can begin implementation. Between any two of these activities, there is a good deal of iteration. Role-playing may send us back to the drawing board 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, while 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.

Table 2: Stages in Designing Multi-Agent Systems

Stage	Focus	Supporting Analysis	Answers
Conceptual Analysis	Components		<ul style="list-style-type: none"> • What system-level behavior do we want? • What kinds of agents might we need to get it? • How should they behave?
Role-Playing	Architecture (Kinematics)	Speech Acts & Dooley Graphs	<ul style="list-style-type: none"> • How do our proposed agents interact with one another in an organization? • What low-level behaviors are needed?
Formal Analysis	Behavior (Dynamics)	<ul style="list-style-type: none"> • Formal Modeling • Simulation & Nonlinear mathematics 	<ul style="list-style-type: none"> • Are the descriptions logically consistent and complete? • What kind of behavior emerges from realistic numbers of agents and interchanges?
Implementation Design	Platforms & Tools		How can I instantiate this design in a deployable system?

4.3.2.2.1 Conceptual Analysis

The specification phase has defined what the system as a whole will do. Conceptual analysis gives us our initial vision of what agents will be involved, and how they will behave. This vision will be refined during later steps. The two major questions to be resolved at this point are the identity and behavior of the agents that will make up the system.

Identify Agents: We have found linguistic case analysis a useful tool in developing the initial partition of the system. The candidate agents identified in this way are then reviewed against design principles that appear to be followed in naturally occurring agent systems.

One widely-used technique for identifying objects in an object-oriented systems analysis [87] is to extract the nouns from a narrative description of the desired system behavior. We use a refinement of this approach, based on linguistic case theory [17, 28, 74]. The basic idea is that 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. The case analysis does not provide a finished system design, but does give a starting point that lends itself to discussion among the developers and has proven to be very robust in terms of surfacing the entities that need to be represented as agents.

To complete the preliminary decomposition, these categories are reviewed and possibly revised against overall system requirements and general principles of agent-based systems. Our laboratory follows a set of principles derived from the analysis of naturally occurring agent systems, since these are demonstrably successful and have proven remarkably robust and adaptable [77].

- *Thing vs. Function:* Classical software engineering techniques condition many systems designers toward “functional decomposition.” This approach is unprecedented in naturally occurring systems, which divide agents on the basis of distinct entities in the physical world rather than functional abstractions. Our experience supports this principle. Each functional agent needs detailed knowledge of many of the physical entities being managed, and so when the physical system changes, the functional agent needs to change as well. However, it is often possible to endow physically defined agents with generic behaviors from which the required functionality will emerge, for widely varying overall populations of agents. 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, 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, thus providing for a wider scope of emergent behavior. (Very roughly, system state space is exponential in the number of agents.) Ecological studies frequently find that the functioning of a biological system depends on minimum population levels much higher than one would suspect based on a naïve analysis of rates of reproduction, predation, and food consumption, because emergent properties essential to the community’s survival are driven by the interaction of many entities. We expect that the same principle will hold true of 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 do not reflect the kind of centralization that often appears in artificial systems. For example, a market economy achieves superior distribution of goods compared with attempts at central economic control. We can hypothesize several reasons for 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 (in the object-oriented sense of the word) for paint-booth agents, but each paint booth should be a separate instantiation 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. The class inheritance mechanisms of the object-oriented platforms on which we construct agents are an excellent support for comparable generalization across the agents we build, but experience shows that the hard part is identifying appropriate generalizations in the first place. Early designs typically multiply differences among agents unnecessarily, while later refinements can make more effective use of the power of inheritance.

Hypothesize Agent Behaviors and Message Types: With a candidate set of agents in hand, we hypothesize their individual behaviors and the classes of messages they can exchange. There exists no algorithm to compute from desired system behaviors to the individual agent behaviors that will yield the system behaviors. Some behaviors (even most) may be straightforward and obvious, but there will always be subsystems where only simulation of example agent behaviors (first in role-playing, later on a computer) can tell us when we have the right behaviors. At this point in our design, our main concern is with identifying 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, and laying the groundwork for role-playing. So we will specify a stochastic process (such as rolling a die or flipping a coin⁶) to choose among internal agent decisions that later will be the subject of detailed computation.

Again, principles observed in naturally occurring systems help us evaluate the agent dynamics and interactions we have proposed.

- *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 that in which their environment changes. Watch out for suggested behaviors that involve extensive up-front planning.
- *Currency:* Naturally occurring multi-agent systems often use some form of currency to achieve global self-organization. The two classical examples are 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. [54] Wherever possible, artificial agent communities should include such a currency.
- *Local Communication:* Agents need to limit the recipients of their messages as much as possible. Wherever possible, instead of "broadcast X," seek to 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.

4.3.2.2.2 Role-Playing

With agents identified and tentative behaviors described, we can experiment with the emergent behavior of selected subsystems by having people play the roles of the various agents. Such a rehearsal does not show the full dynamic behavior that would be 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.

Select Subsystems and Scripts for Role Playing: In our experience, a large proportion of the individual behaviors for most of the agents will be fairly obvious. This empirical result is fortunate, since role-playing a complete system

⁶ By treating a penny, a dime, a nickel, and a quarter as successively higher bit positions, up to 2^4 alternatives can be represented.

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, we focus on subsystems of a dozen or so agents where we are least comfortable about the match between individual and system behaviors.

In addition to selecting these subsystems for role-playing, we need several scripts of the desired system behavior. For example, if we seek a system with homeostasis, we need to identify 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.

Assign Agents to People: 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, it may be necessary for a single person to handle a complete class of agents. In this case we need to 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.

As noted in the introduction to this chapter, the environment is not necessarily passive, but may have state and processes associated with it. It differs from an agent in that it is unbounded. In addition to the agents proposed for the system being engineered (the "system agents"), a person should be assigned to play the role of the environment manipulated by the system. The environment 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 the 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.

Record Actions: To support later analysis, we capture all actions that agents (both system and environment) take external to themselves. 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:

- The identity of the sending agent
- The identity of the receiving agent
- The time the card is sent
- The identity of the agent whose card stimulated this one
- 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 are a useful way to determine 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 view of all participants, it is easy to maintain an unambiguous ordering of the cards that permits reconstruction of the conversation.

Facilitate the Interactions among Agents: A facilitator who is not one of the agents should oversee the execution of each script. There are three phases in this responsibility.

- *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.

- *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.
- *Debrief:* After completion of 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?

Graph the Conversations: Enhanced Dooley Graphs [75] are a useful tool to 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 takes on different roles in the course of the conversation. These roles are good candidates for 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.

4.3.2.2.3 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 [23] is one example among many of this approach. Because of the complexity and resulting cost of the “correctness proof” approach to program development, it is applied only selectively for the development of conventional software, and it is to be expected that formal methods will similarly be selectively deployed.

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 limited role-play scales up to a full population of agents. 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.

4.3.2.2.4 Implementation Design

The transition from design to implementation is the selection of 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.

4.3.3 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.

Each case description in this section offers a description organized around the natural features of the tool, rather than attempting to conform to a general taxonomy.

4.3.3.1 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, which are often being sold with a PC-based development environment that permits programming in a high-level language. The single-chip Basic Stamp [72] has nonvolatile storage for 500-600 lines of Basic code, which it interprets at about 4000 instructions per second. It includes 18 I/O pins and two dedicated serial lines. Z-World [98] offers a line of C-programmable single-board microcomputers. The basic model offers a 9MHz Z180, four serial ports, eight high-current outputs, seven analog inputs and one analog output. Computers such as these require additional peripheral support for inter-agent communication over any significant distance.

The next level of sophistication is represented by LonWorks [22], which is built around a single-chip computer that includes the LonTalk protocol, 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) [26]. The PIM is 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. Each cell has fixed and guaranteed resources, regardless of the total number of cells in the system. The cells are organized in a rectangular array and execute synchronously, 60 times a second. Unlike most MAS architectures but like many naturally occurring agent systems, PIM cells communicate through shared environmental data rather than messages. The PIM is programmed in Paracell, a high-level rule-based language that associates a module of code (called a "tile") with each cell. The Navigator groups tiles in a hierarchical structure that developers can easily traverse through a graphic interface. Its rich development environment, I/O capabilities, and real-time response make the PIM a powerful prototyping environment for agent-based control problems.

4.3.3.2 Standards

The emergence of broadly accepted standards helps bring users and developers together into a critical mass. Industrial users depend on others to provide tools and methods for agent development. Developers of tools and methods need to be assured that buyers will be available for the tools they produce. 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, potential 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) [65] 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 [14], the Foundation for Intelligent Physical Agents, is a world-wide consortial effort devoted to agent standards in general.

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

Physically moving bytes around can be problematic in an industrial environment. Devices such as arc welders, induction furnaces, and motor starters can generate high levels of electromagnetic interference that quickly overwhelm many network structures that are completely adequate for laboratory work. Until recently, industrial control relied on point-to-point wiring of I/O points on a controller to the devices being controlled, requiring separate conductors for each device. Control-area networks replace these unwieldy tangles of wire with multiplexed communications, and are engineered to cope with potential interference. DeviceNet [69] is a standard for control-area networks that was originally devised by Rockwell Automation/Allen-Bradley, and is now independently maintained. While the initial configuration of DeviceNet favors hierarchically structured master-slave

architectures, the standard has the latitude to support peer-to-peer interactions, and is an important candidate for widespread deployment of agents in control environments.

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 [70], 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. KIF [35], the Knowledge Interchange Format, is a language for exchanging knowledge among agents. KIF offers declarative semantics, equivalence to first-order logic, and the ability to express knowledge about knowledge, and is being developed into an American National Standard (ANSI). KQML [29], the Knowledge Query and Manipulation Language, is a language that agents can use to communicate their attitudes toward information. These attitudes (such as querying, asserting, or offering) are the domain of speech act theory, a branch of linguistics that explores utterances as actions that can succeed or fail, not just as propositions that may be true or false. KQML is the most widely-used effort to apply speech act theory to interagent communication. KQML and KIF are complementary: 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 may need to be collocated on the same processor. A part agent may need to move from one machine agent to another during its residency in the shop. Java [89] is a standard mechanism to enable 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.

The examples of standards considered so far all 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, Grafset [8] 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 [27], and Grafset enjoys widespread industrial use as a representation for control logic, making it a good candidate for implementing industrial agents. Gensym's Agent Development Environment (described later in this section) uses Grafset as the main interface for defining agent behaviors.

4.3.3.3 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 the availability of 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 one of the major roadblocks to wider agent deployment in industry [76], and within the past year a number of products have emerged to address this need. 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.

4.3.3.3.1 An Agent as a Single Reactive Process

IBM's Agent Building Environment [42] 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. The major orientation of ABE is toward isolated intelligent agents, rather than multi-agent systems.

4.3.3.3.2 Agents as Capitalists

Dissipative mechanisms such as currency flows in markets are a powerful way to achieve coordination in a decentralized system. Agorics is a software development company that specializes in applying 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.

4.3.3.3.3 Agents as Travelers

One widely used definition of an agent is a process that 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, [92, 93] which models the world as places (processors) and agents (processes). The basic concepts are documented in General Magic’s patent on mobile agents [94].

With the growth of Java as the lingua franca of the Internet, 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™ [43] 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™ [34]. ObjectSpace’s Voyager™ product [68] provides a Java-based Object Request Broker (ORB) designed for mobile agents. ObjectSpace offers a detailed comparison of Aglets™, Odyssey™, and Voyager™ [67]. All three tools are available free for noncommercial use, and Voyager™ is freely available for commercial use as well.

4.3.3.3.4 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) [36] 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 Grafset standard for industrial-strength Petri nets, and a simulation tool for performing simulations of (distributed) agent-based applications. The current version of ADE is available for Gensym customers. Gensym is evaluating ADE to support it as a product.

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.

An agent can be involved in more than one activity at a time. For example, an agent representing a machine tool might concurrently be machining one part and engaging in negotiations for a future job. An ADE agent can have multiple contexts, each supporting a single activity. Thus an agent can be multi-threaded, with each context representing a single thread. Handling of messages and other events is always done within a particular context. The context maintains the state of its associated activity in the agent, and has a message handler that deals with messages appropriate to that context. Message handlers are written either in G2, or in Grafacet using a graphical interface.

Because agent-based systems can exhibit complex emergent dynamics, simulation is an essential component of development. ADE supports integration of simulation in development by providing a simulation agent class that can be used to construct discrete event simulations of external devices or processes. During development, the application agents interact with these simulation agents, and then are switched over to the physical devices when the system is deployed. Currently, ADE is not intended to be a robust deployment environment, though the underlying G2 is regularly used in real-time control applications and one may reasonably expect ADE to move in that direction.

In support of the AARIA project described earlier in this chapter, Industrial Automation, Inc. (IAI), the AARIA prime contractor, created the first version of Cybele, a NeXT-based agent infrastructure. [25] Based on the requirements analyzed in [56], 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 [58] 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 blackboard. 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 on which the Metra scheduling system described earlier is constructed.

4.3.3.3.5 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 often used in building 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: *dMARS* [37], 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 has five components:

- 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.

dMARS supports agent communities running on networks of UNIXTM machines. The system is written in C++, and provides facilities for wrapping existing C++ modules into pseudo-agents for integration into the overall dMARS community.

D-Muse: D-Muse [97] 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. The main mode of interaction is through mirrored objects, a capability that permits one agent (the publisher) to create 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.

ABS: 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*, the lowest of the four, 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. It supports standard knowledge-representation methods including nonmonotonic reasoning, deductive reasoning, inconsistency detection, automated concept classification, subsumption-based theorem proving, and truth maintenance.

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* provides supports two categories of information 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.

4.3.4 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: As discussed earlier in this chapter, one of the great benefits of an agent architecture is its 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 tradespeople who are not 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 will reasonably 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.

5. Conclusion

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 implementation technology is, nor what one has learned about that technology, but how well the entire project 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. Over time, one learns that 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 offers 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 described in the last section 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 Multi-Agent Technology) conferences [91] 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 seek to 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 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.

6. 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/>, 1996.
- [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] E. Bierel and J.-M. Roussel. "Welcome to the GRAFCET Home Page". <http://www.lurpa.ens-cachan.fr/grafcet.html>, 1995.
- [9] R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1 (March)):14-23, 1986.
- [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] B. Burmeister. Models and Methodology for Agent-Oriented Analysis and Design. In *Proceedings of Workshop on Agent-Oriented Programming and Distributed Systems, KI'96 (September)*, 1996.
- [12] 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.
- [13] F.-C. Cheong. *Internet Agents: Spiders, Wanderers, Brokers, and Bots*. Indianapolis, IN, New Riders, 1996.
- [14] L. Chiariglione. Foundation for Intelligent Physical Agents. <http://drogo.cselt.stet.it/fipa/>, 1987.
- [15] K. T. Chung and C.-H. Wu. Dynamic Scheduling with Intelligent Agents: An Application Note. Metra Application Note 105, Metra, Palo Alto, CA, 1997.
- [16] S. H. Clearwater, Editor. *Market-Based Control: A Paradigm for Distributed Resource Allocation*. Singapore, World Scientific, 1996.
- [17] W. A. Cook. *Case Grammar: Development of the Matrix Model*. Washington, Georgetown University, 1979.
- [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] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63-109, 1983.
- [21] 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.
- [22] Echelon. Welcome to Echelon. <http://www.lonworks.echelon.com>, 1997.
- [23] P. v. Eck. The DESIRE Research Programme. <http://www.cs.vu.nl/vakgroepen/ai/projects/desire/>, 1996.
- [24] G. Ekberg. Benefits of Autonomous Agent Approach to Manufacturing Systems Control. In *Proceedings of Third Annual Chaos East Technical Conference*, R. Morley, Inc., 1997.
- [25] K. Erol. Cybele: An Infrastructure for Autonomous Agents. <http://www.i-a-i.com/projects/cybele/index.html>, 1996.
- [26] P. Fandel, R. DeSimone, and H. Mitchell. Paracell/PIM Product Summary Preface. <http://www.flavors.com/docn/ppps/index.html>, 1996.
- [27] J. Ferber. *Les systèmes multi-agents: vers une intelligence collective*. Paris, France, InterEditions, 1995.
- [28] C. J. Fillmore. The Case for Case Reopened. *Studies in Syntax and Semantics*, (8):59-81, 1977.
- [29] T. Finin. UMBC KQML Web. <http://www.cs.umbc.edu/kqml/>, 1997.
- [30] 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.
- [31] 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.
- [32] M. S. Fox. The Integrated Supply Chain Management Project. <http://www.ie.utoronto.ca/EIL/iscm-descr.html>, 1996.
- [33] 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.
- [34] General Magic. General Magic: Odyssey. <http://www.genmagic.com/agents/odyssey.html>, 1997.
- [35] M. R. Genesereth. Knowledge Interchange Format (KIF). <http://logic.stanford.edu/kif/>, 1996.
- [36] Gensym. Agent Development Environment (ADE) Overview. Gensym, Inc., Cambridge, MA, 1997.
- [37] M. Georgeff. dMARS Technical Overview. http://www.aaii.oz.au/proj/dmars_tech_overview/dMARS-1.html, 1996.
- [38] J. Hatvany. Intelligence and Cooperation in Heterarchic Manufacturing Systems. *Robotics & Computer-Integrated Manufacturing*, 2(2):101-104, 1985.
- [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] HP. The JetSend Protocol. <http://www.jetsend.hp.com/Whitepaper.html>, 1997.
- [42] IBM. IBM Agent Building Environment (ABE) - A toolkit for building intelligent agent applications. <http://www.networking.ibm.com/iag/iagsoft.htm>, 1997.
- [43] IBM. IBM Aglets Workbench--Home Page. <http://www.trl.ibm.co.jp/aglets/>, 1997.
- [44] H. Ihara and K. Mori. Autonomous Decentralized Computer Control Systems. *IEEE Computer*, 17(8):57-66, 1984.

- [45] N. Jennings. Applying Agent Technology. Plenary presentation at PAAM'96. 1996.
- [46] N. R. Jennings. *Cooperation in Industrial Multi-Agent Systems*. 1994.
- [47] N. R. Jennings. The ARCHON System. http://www.elec.qmw.ac.uk/dai/archon/test_1.html, 1996.
- [48] N. R. Jennings. ADEPT: Advanced Decision Environment for Process Tasks. <http://www.elec.qmw.ac.uk/dai/projects/adept/>, 1997.
- [49] N. R. Jennings and J. R. Campos. Towards a Social Level Characterisation of Socially Responsible Agents. *IEE Proceedings Software Engineering*, 144(no 1):11-25, 1997.
- [50] 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>.
- [51] 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.
- [52] N. R. Jennings, E. H. Mamdani, I. Laresgoiti, J. Perez, and J. Corera. GRATE: A General Framework for Co-operative Problem Solving. *IEE-BCS Journal of Intelligent Systems Engineering*, 1(2 (Winter)):102-14, 1992.
- [53] M. Koster. The Web Robots Pages. <http://info.webcrawler.com/mak/projects/robots/robots.html>, 1996.
- [54] P. N. Kugler and M. T. Turvey. *Information, Natural Law, and the Self-Assembly of Rhythmic Movement*. Lawrence Erlbaum, 1987.
- [55] H. S. Lee, S. Murthy, S. W. Haider, and D. Morse. Primary Production Scheduling at Steel-Making Industries. *IBM Journal of Research and Development*, 40(2 (March)):231-252, 1996.
- [56] 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.
- [57] J. Maley. Managing the Flow of Intelligent Parts. *Robotics and Computer-Integrated Manufacturing*, 4(3/4):525-30, 1988.
- [58] Metra. Agent Technology: UNITY_Agent: An Agent Enabler. Metra Corporation, San Jose, CA, 1997.
- [59] 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.
- [60] 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.
- [61] J. P. Müller. *The Design of Intelligent Agents*. Berlin, Springer, 1996.
- [62] S. Murthy, R. Akkiraju, J. Rachlin, and F. Wu. Agent-Based Cooperative Scheduling. In *Proceedings of AAAI Workshop on Constraints and Agents*, pages 112-117, AAAI Press, 1997.
- [63] R. N. Nagel and R. Dove. *21st Century Manufacturing Enterprise Strategy*. Bethlehem, PA, Agility Forum, 1991.
- [64] A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87-127, 1982.
- [65] NIIIP. Mother NIIIP Homepage. <http://www.niiip.org>, 1996.
- [66] 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.

- [67] ObjectSpace. ObjectSpace Voyager, General Magic Odyssey, IBM Aglets: A Comparison. <http://www.objectspace.com/voyager/VoyagerAgentComparisons.PDF>, ObjectSpace, Inc., Dallas, TX, 1997.
- [68] ObjectSpace. Voyager(tm) Core Package Technical Overview. <http://www.objectspace.com/Voyager/VoyagerTechOviewOnlineVersion.PDF>, ObjectSpace, Inc., Dallas, TX, 1997.
- [69] ODVA. Learn About DeviceNet. <http://www.industry.net/c/orgunpro/odva/dev-net>, 1997.
- [70] OMG. The Common Object Request Broker: Architecture and Specification, Revision 2. OMG Technical Document formal/97-02-25, <http://www.omg.org/corba/corbiop.htm>, Object Management Group, 1996.
- [71] 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.
- [72] Parallax. BASIC Stamp FAQs. ftp://ftp.parallaxinc.com/pub/acrobat/stamp_faqs.pdf, Parallax, Inc., 1997.
- [73] H. V. D. Parunak. Manufacturing Experience with the Contract Net. In M. N. Huhns, Editor, *Distributed Artificial Intelligence*, pages 285-310. Pitman, London, 1987.
- [74] 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.
- [75] 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.
- [76] H. V. D. Parunak. Workshop Report: Implementing Manufacturing Agents. In *Proceedings of PAAM'96*, 1996.
- [77] 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>.
- [78] 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.
- [79] 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.
- [80] H. V. D. Parunak, J. Sauter, and S. J. Clark. Toward the Specification and Design of Industrial Synthetic Ecosystems. In *Proceedings of Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, Springer, 1997.
- [81] 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.
- [82] 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.
- [83] J.-F. Perrot. Preface. In J. Ferber, Editor, *Les Systèmes Multi-Agents: Vers une intelligence collective*, pages xiii-xiv. InterEditions, Paris, 1995.
- [84] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI Architecture. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR-91)*, pages 473-484, Morgan Kaufman, 1991.

- [85] R. Roberts. *Zone Logic: A Unique Method of Practical Artificial Intelligence*. Radnor, PA, Compute! Books, 1989.
- [86] G. C. Roman. A Taxonomy of Current Issues in Requirements Engineering. *IEEE Computer*, (April):14-22, 1985.
- [87] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, Prentice Hall, 1991.
- [88] 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.
- [89] Sun. JavaSoft Home Page. <http://java.sun.com>, 1997.
- [90] 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.
- [91] TPAC. The Practical Application Company. <http://www.demon.co.uk/ar/TPAC/index.html>, 1997.
- [92] J. E. White. Telescript Technology: The Foundation for the Electronic Marketplace. 1994.
- [93] J. E. White. Telescript: Transportable Agent Systems. <http://www.genmagic.com/Telescript/>, 1996.
- [94] 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.
- [95] T. Wittig. *ARCHON: An Architecture for Multi-agent Systems*. New York, Ellis Horwood, 1992.
- [96] F. Ygge and H. Akkermans. Making a Case for Multi-Agent Systems. In *Proceedings of MAAMAW'97*, 1997.
- [97] R. Zanconato. An Inter-Agent Communication Model for Real-Time Distributed AI Applications. In *Proceedings of PAAM-96: The First International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 755-771, The Practical Application Company Ltd, 1996.
- [98] Z-World. About Z-World. <http://www.zworld.com/about.html>, 1997.