

# Software Development with Objects, Agents, and Services

Michael N. Huhns

University of South Carolina  
Department of Computer Science and Engineering  
Columbia, SC 29208 USA  
Huhns@sc.edu  
<http://www.cse.sc.edu/huhns>

**Abstract.** Software system developers have a number of methodologies available to them. Each is based on a set of abstractions and each has several advantages and disadvantages. This paper surveys the available methodologies and evaluates them with regard to their support for reuse and robustness. It focuses on the trends that are evident in software engineering and the impact that the trends will have on system development. It concludes that both current trends and anticipated needs converge on a multiagent-based service-oriented computing methodology.

## 1 Introduction

Computing is in the midst of a paradigm shift. After decades of progress on representations and algorithms geared toward individual computations, the emphasis is shifting toward *interactions* among computations. The motivation is practical, but there are major theoretical implications. Current techniques are inadequate for applications such as ubiquitous information access, electronic commerce, and digital libraries, which involve a number of independently designed and operated subsystems. The metaphor of interaction emphasizes the autonomy of computations and their ability to interface with each other and their environment. Therefore, it can be a powerful conceptual basis for designing solutions for the above applications.

The paradigm shift is being manifested by trends towards both service-oriented computing and multiagent-based systems. These trends are complementary: services need the ability to behave proactively and to negotiate as provided by agents, while agents need the capability for mutual understanding as being provided by the Semantic Web in the form of ontologies for objects and processes.

There are several methodologies for developing software that might be suitable for large distributed applications. There are many requirements for a methodology, but it must support distributed development and execution, software reuse, and robustness.

## 2 Fundamental Abstractions

Each programming methodology has an associated set of abstractions that characterize its approach and that determine its appropriate applications. The abstractions help in elucidating the benefits and limitations of a methodology. This section describes the abstractions for the methodologies and evaluates them with regard to software engineering's two major goals: reuse and robustness.

### 2.1 Procedural Programming

The earliest programming languages, exemplified by Fortran and COBOL, were procedural. The major abstraction supported by procedural programming is the notion of an algorithm and its representation in instructions suitable for a von Neumann architecture. Procedural programs are typically intended to be executed discretely in a batch mode with a specific start and end. An advantage of procedural programming is that computational efficiency is more easily optimized, because representations match the computing architecture. For this same reason, compilers are then easier to construct.

Early computer systems had little memory and slow CPUs, so it was natural to design programs that were close to the computer systems that would execute them. However, this resulted in several limitations for procedural programming: it does not scale up for large applications, it is difficult to distribute either its development or its execution, it is difficult to reuse the code for new applications, and it obscures errors and bugs, making it difficult for developers to locate and repair them.

### 2.2 Object-Oriented Programming

The major abstractions supported by object-oriented programming are representations of both real-world entities and programming entities. For example, an object might be a *truck* or a *random-number generator*. The representations include the properties and the procedures of the objects. The benefits of OOP are achieved through encapsulation, polymorphism, and inheritance. Encapsulation enables and encourages software to be developed and tested in a modular fashion. Inheritance reuses code automatically, thereby improving both robustness and ease of development. Polymorphism makes it easier for developers to construct modules independently.

The disadvantages of object-oriented programming are related to its lack of abstractions or support for the relationships and interactions among objects. Complex applications typically require many objects to interoperate coherently. When an object is designed and created, it is not always known how and when its functionality will be used and its state will be affected. This can lead to unanticipated errors. Furthermore, by being passive in the sense that an object's functionality must be invoked externally, it is not an accurate model of real-world entities as desired. The abstractions in object-oriented programming are not wrong, they are simply insufficient.

### 2.3 Component-Based Design and Development

Components are also objects, but they are typically larger both horizontally (having more features) and vertically (can extend from user interfaces to operating system hooks). They also introduce an additional abstraction: a self-description facility. This is important, because it enables objects to find each other and to locate the external functionality that they wish to invoke. More often, it enables developers to locate the components they wish to reuse, thereby easing the development effort—a major advantage.

A disadvantage is that the self-description abstraction does not go far enough: components do not do anything to aid in their reuse or in their being discovered at runtime. Like objects, they are essentially passive, waiting to be discovered and utilized.

### 2.4 Agent-Oriented Programming

The abstractions provided by agents are derived from psychology and from animate behavior. From psychology, the abstractions are a mentalistic state consisting of beliefs, desires, and intentions. From animate behavior, the abstractions are active functionality and autonomy. Combined, activity and autonomy enable an agent to guard against unwarranted changes to its internal state and unwanted uses of its functionality, as well as appropriate participation in an environment.

Because of a focus on the individual, i.e., one agent at a time, the resultant systems tend to consist of complex entities that use only simple interactions to deal with a complex environment. The programming effort is thus concentrated on developing the complex entities and providing them with the knowledge and beliefs needed to act intelligently in their environment, i.e., choose their intentions so that they achieve their desires. There is little effort on modeling the environment and incorporating that model into the agent.

### 2.5 Multiagent-Oriented Programming

Software development based on multiagent systems takes advantage of abstractions derived from sociology, economy, and law. The abstractions are social, consisting of commitments and obligations. The focus is on the interactions among the active entities (i.e., agents) and the passive entities (i.e., other objects) in an environment. The development effort focuses on modeling the environment and, more importantly, the other agents, leading to mutual beliefs, joint intentions, and common goals (desires).

Commitments provide a natural abstraction to encode relationships among autonomous, heterogeneous parties. Commitments are important for organizational control, because they provide a layer that mediates across the declarative semantics of organizations from the operational behavior of the team members. Their advantages to organizational control are:

- Commitments can be assigned to roles, so that any agent that fills the role will inherit the role’s commitments
- Commitments can be delegated
- Commitments can be reassigned. For example, if an agent fails to meet its commitment, then the responsible authority (another agent) can release the first agent from the commitment and delegate it to another
- Commitments can be negotiated. One agent might ask another (a peer) to take over a commitment that could not otherwise be met
- Commitments can fail to be met; the failure can then be communicated to an agent with the authority to release the original commitment and reassign it.

Obligations, i.e., commitments to follow required policies, enable an agent to reason about the relationship between its responsibilities and its decision-making constraints and goals. This feature decides which organizational policies apply for the current situation and marks as unacceptable any intended actions that are inappropriate.

This paradigm is ideally suited for addressing the disadvantages inherent in the earlier paradigms while meeting the objectives of open enterprise-wide applications. However, it requires a large population of agents that have a mutual understanding of requirements and capabilities. Service-oriented computing, described in Section 4, removes this disadvantage by providing generators and sources for agents.

Table 1 summarizes the major features of existing software paradigms, and the features promised by the service-oriented and multiagent-based approaches described below.

**Table 1.** Features of Programming Languages and Paradigms

<b>Concept</b>	<b>Procedural Language</b>	<b>Object Language</b>	<b>Multiagent-Based Service-Oriented Language</b>
<b>Abstraction</b>	Type	Class	Service/Interaction
<b>Building Block</b>	Instance, Data	Object	Agent
<b>Computation Model</b>	Procedure/Call	Method/Message	Perceive/Reason/Act
<b>Design Paradigm Architecture</b>	Tree of Procedures Functional Decomposition	Interaction Patterns Inheritance and Polymorphism	Cooperative Interaction Managers, Assistants, and Peers
<b>Modes of Behavior</b>	Coding	Designing and Using	Enabling and Enacting
<b>Terminology</b>	Implement	Engineer	Activate

### 3 Assessment

We focus on how to build software out of agents, i.e., agents as the fundamental building blocks for software, rather than how to engineer agents *per se* or how to use agents to help construct software, i.e., agent-based CASE tools.

The conventional computer science viewpoint is that the behavior of a module must be understood before it is executed, whether it is invoked by another module or by a human. Such understanding is not practical when

- The execution environment is open
- The modules are complex
- The modules are numerous
- The modules can learn or adapt

Unfortunately, modern enterprise-level applications are characterized by all of these. It is such applications that service-oriented computing is intended to address. It leads to the potentially feasible achievement of programming-in-the-large.

The procedural and declarative approaches to programming suffer from being primarily line-at-a-time techniques, with a basis in functional decomposition. Object technology improves these by replacing decomposition with inheritance hierarchies and polymorphisms. It enables design reuse of larger patterns and components [3]. However, inheritance and polymorphism are just as complex and error prone as decomposition, and the great complexity of interactions among objects limits their production and use to a small community of software engineers. By focusing on encapsulating data structures into objects and the relationships among objects, it supports a data-centric view that makes it difficult to think about sequences of activity and dataflow. Scenarios overcome this difficulty by depicting message sequences and threads of control, but they are not well supported by current object languages.

### 4 Service-Oriented Computing

Service-oriented computing relies on standardized Web services. The current incarnation of Web services emphasizes a single provider offering a single service to a single requester. This is in keeping with a client-server architectural view of the Web.

Service-oriented computing provides two major benefits. One, it enables new kinds of flexible business applications of open systems that simply would not be possible otherwise. When new techniques improve the reaction times of organizations and people from weeks to seconds, they change the very structure of business. This is not a mere quantitative change, but a major qualitative change. It has ramifications on how business is conducted. Two, service-oriented computing improves the productivity of programming and administering applications in open systems. Such applications are notoriously complex. By offering productivity gains, new techniques make the above kinds of applications practical, thus helping bring them to fruition.

Additional benefits of service-oriented computing are the following:

- To achieve the interoperation of applications within an enterprise, it provides the tools to model the information and relate the models, construct processes over the systems, assert and guarantee transactional properties, add in flexible decision-support, and relate the functioning of the component software systems to the organizations that they represent.
- For interenterprise interoperation, it additionally provides the ability for the interacting parties to choreograph their behaviors so that each may apply its local policies autonomously and yet achieve effective and coherent cross-enterprise processes.
- It enables the customization of new applications by providing a Web service interface that eliminates messaging problems and by providing a semantic basis to customize the functioning of the application.
- It enables dynamic selection of business partners based on quality-of-service criteria that each party can customize for itself.
- It enables the efficient usage of Grid resources.
- It facilitates utility computing, especially where redundant services can be used to achieve fault tolerance.
- It provides a semantically rich and flexible computational model that simplifies software development.

To realize the above advantages, SOAs impose the following requirements:

**Loose coupling** No tight transactional properties would generally apply among the components. In general, it would not be appropriate to specify the consistency of data across the information resources that are parts of the various components. However, it would be reasonable to think of the high-level contractual relationships through which the interactions among the components are specified.

**Implementation neutrality** The interface is what matters. We cannot depend on the details of the implementations of the interacting components. In particular, the approach cannot be specific to a set of programming languages.

**Flexible configurability** The system is configured late and flexibly. In other words, the different components are bound to each other late in the process. The configuration can change dynamically.

**Long lifetime** We do not necessarily advocate a long lifetime for our components. However, since we are dealing with computations among autonomous heterogeneous parties in dynamic environments, we must always be able to handle exceptions. This means that the components must exist long enough to be able to detect any relevant exceptions, to take corrective action, and to respond to the corrective actions taken by others. Components must exist long enough to be discovered, to be relied upon, and to engender trust in their behavior.

**Granularity** The participants in an SOA should be understood at a coarse granularity. That is, instead of modeling actions and interactions at a detailed level, it would be better to capture the essential high-level qualities

that are (or should be) visible for the purposes of business contracts among the participants. Coarse granularity reduces dependencies among the participants and reduces communications to a few messages of greater significance.

**Teams** Instead of framing computations centrally, it would be better to think in terms of how computations are realized by autonomous parties. In other words, instead of a participant commanding its partners, computation in open systems is more a matter of business partners working as a team. That is, instead of an individual, a team of cooperating participants is a better modeling unit. A team-oriented view is a consequence of taking a peer-to-peer architecture seriously.

Researchers in multiagent systems (MAS) confronted the challenges of open systems early on when they attempted to develop autonomous agents that would solve problems cooperatively, or compete intelligently. Thus, ideas similar to service-oriented architectures were developed in the MAS literature. Although SOAs might not be brand new, they address the fundamental challenges of open systems. Clearly the time is right for such architectures to become more prevalent. What service-oriented computing adds to MAS ideas is the ability to build on conventional information technology and do so in a standardized manner so that tools can facilitate the practical development of large-scale systems.

## 5 Engineering SOC Applications

Constructing an application by composing services first requires that existing services, with the functionalities they provide, be identified. Where essential services are missing, they must be constructed by the application developer or their construction out-sourced. The next step is to select, plan, or specify the desired combination of services. Finally, the composition of services is executed and monitored for success or faults.

Current approaches take a procedural view of service composition by formulating workflow graphs that can be stepped through in a simple manner. Because of this, the main engineering challenges that arise concern standardizing on the data, e.g., through syntax or the semantics. However, Web services have attributes that set them apart from traditional closed applications: they are autonomous, heterogeneous, long-lived, and interact in subtle ways to cooperate or compete. Engineering a composition of such services requires abstractions and tools that bring these essential attributes to the fore. When the requirements are expressive, they highlight the potential violations, e.g., the failure modes of a composition.

Engineering composed services thus requires capturing patterns of semantic and pragmatic constraints on how the services may participate in different compositions. It also requires tools to help reject unsuitable compositions so that only acceptable systems are built.

A key aspect in the design of a composed service or a multiagent system is to maintain global coherence, often without explicit global control. This calls

for a means to pool knowledge and evidence, determine shared goals, determine common tasks across services, and avoid unnecessary conflicts. The result is a collaboration.

Several challenges regarding transport, messaging, and security constraints must be handled for each collaboration. In general, business collaborations are becoming increasingly complex, and most systems will be dealing with multiple collaborations at the same time. If the transactions are legally bound or even otherwise, a nonrepudiation condition may have to be satisfied. Lastly, as usual, there is always a possibility of exceptions.

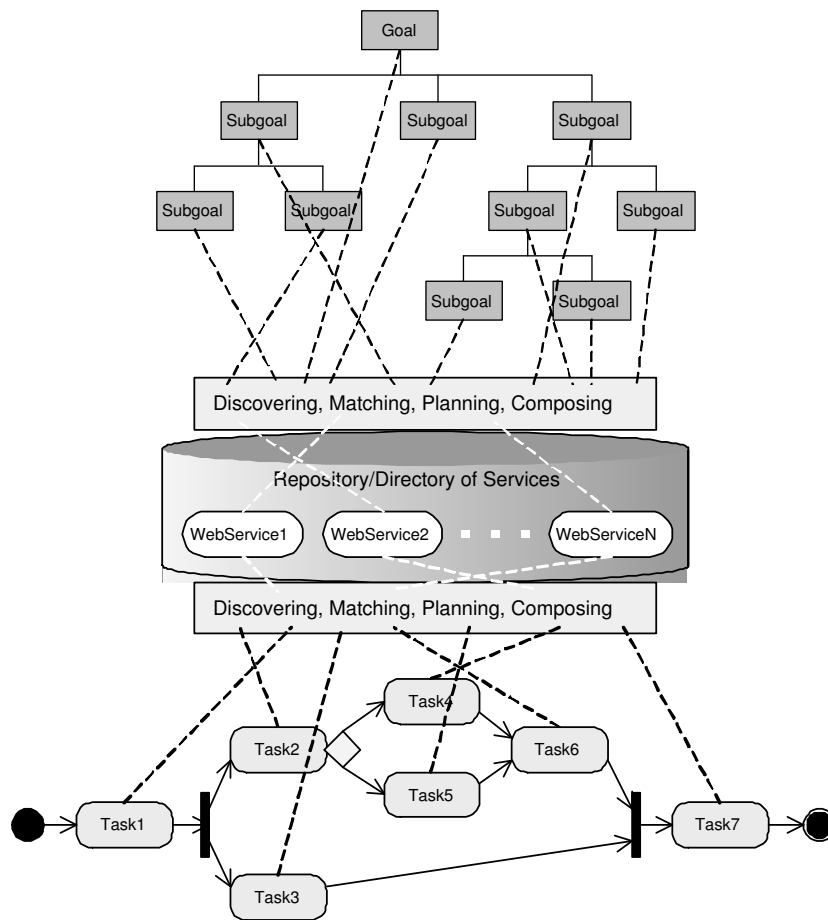
Automating business transactions using ontologies and agents is an appealing approach to meeting the above challenges. Steps toward such technologies are already underway in industry. For example, OAG, OASIS, bizTalk, and RosettaNet are standardizing the syntax and semantics of the *documents* that are exchanged during a B2B transaction. What is also needed is a basis for standardizing (and automating) the *behaviors* that are expected of the participants in a B2B transaction. Further, *misbehaviors* must be handled. For example, a precise specification of a purchase order does not say what to do in the following case: if a company does not receive a response, should it assume that the recipient was not interested or that the PO was lost?

Whether an architecture for an application is specified in terms of a workflow, causal model, process model, goal-subgoal graph, or some other modeling formalism, it must be realized by compositions of available services. These services might be found in a local repository or located across the Internet. Engineering a service-oriented computing system is thus a process of discovering and matching the appropriate services, planning their interactions, and composing them at run-time. This process is illustrated in Figure 1. Also note that when deciding on tasks, or decomposing goals into subgoals, it is important to end up with tasks and goals that match services available in a repository or ones that can be constructed easily. An unresolved problem is that the services within the repository are typically not organized.

## 6 Consensus Software

The two most needed research advancements for service-oriented computing are to investigate how mutual understanding can be reached among autonomous entities and how consensus behavior among agents with overlapping interests can be achieved. We are investigating these within the context of constructing conventional software out of agent-based components.

We first hypothesize that robust software can be achieved through redundancy, where the redundancy is achieved by agents that have different algorithms but similar responsibilities. The agents are produced by wrapping conventional algorithms with a minimal set of agent capabilities, which we specify. Our initial experiments in verifying our hypothesis show an improvement in robustness due to redundancy. We then advance the hypothesis that the redundant algorithms can be obtained through the efforts of many people outside of the traditional soft-



**Fig. 1.** Engineering an SOC system is a process of discovering and composing the proper services to satisfy a specification, whether the specification is expressed in terms of a goal graph (top), a workflow (bottom), or some other model

ware development community. To be successful, such widespread development will require tools, an infrastructure, and a means for semantic reconciliation, and we investigate the characteristics and requirements for these. The paper analyzes the relationship between the software systems that manage and control societal institutions and the members of the society.

First, consensus software can be internally coherent and comprehensible, because it can make use of the consensus ontologies we are developing.

Second, a software-component industry is arising that will distribute on demand components that have functionality customized to a user's needs [44]. However, because of this uniqueness, how can component providers be confident that their components will behave properly? This is a problem that can be solved by agent-based components that actively cooperate with other components to realize system requirements or a user's goals.

Because each application executes as a set of geographically distributed parts, a distributed active-object architecture is required. It is likely that the objects taking part in the application were developed in various languages and execute on various hardware platforms. A simple, powerful paradigm is needed for communications among these heterogeneous objects. We anticipate that agent-based Web services can fulfill this requirement.

Because the identities of the resources are not known when the application is developed, there must be an infrastructure to enable the discovery of pertinent objects. Once an object has been discovered, the infrastructure must facilitate the establishment of constructive communication between the new object and existing objects in the application, which can be satisfied by our agent-based Web services.

Because the pattern of interaction among the objects is a critical part of the application and may vary over time, it is important that this pattern (work-flow) be explicitly represented and available to both the application and the user. When an object has been discovered to be relevant to an application, the language for interaction and the pattern of interaction with the object must be determined. This interaction becomes part of the larger set of object interactions that make up the application. The objects collaborate with each other to carry out the application task.

## 6.1 The Relationship of Software Systems to Social Institutions

One consequence of object-oriented programming is that computer software more closely resembles and models the real world. Non-OOP paradigms, such as procedural programming methodologies, typically model mathematical abstractions. (For example, a FORTRAN subroutine might compute a matrix inverse, whereas an OOP module might represent the class Automobile.) Although OOP-developed software mimics the world, it is not developed in this way. That is, the software is produced by a small number of professional developers, rather than by the "world." For example, when I visit Amazon.com, an agent with a model of me makes suggestions about purchases I might like to make, but if the model is inaccurate there is no way for me to improve it.

As another example, my employer has a software and information model of me as an instance of an Employee class, but I did not contribute directly or consciously to the construction of that model. Instead, the models of me and all other employees in my organization were constructed centrally. There is always a tension between the order that comes with centralization and the freedom that comes with decentralization. This is reflected in similar debates concerning privacy versus security, prevention versus protection, and, more generally, free markets versus centralized economies.

The economic historian Douglass North believed that institutions evolve toward free markets, where institutions are conceptual structures that coordinate the activities of people [23]. Institutions, in his view, consist of a network of relationships, including all of the skills, strategies, and norms that the participants contribute to the institution. He believed that the driving force behind the evolution of institutions was self-interest.

During the current economic downturn, many organizations are facing declines in their revenues and budgets. A common, centralized response is an across-the-board cut in salaries and expenses, with a view that each employee should receive a minimum salary. An employee's individual view is that he should receive the maximum salary and the deficit should be made up for elsewhere. Not everyone can receive a maximum salary, however, because the institution would fail and no one would receive any salary. But there is pressure for the institution to grow so that everyone will receive more.

In contrast, the economist John Commons viewed an institution as a set of working rules that govern an individual's behavior [6]. The rules codify culture and practices and are defined by collective bargaining. As a result, institutions evolve ideally towards democracy. For the above example, according to Commons's view, each Employee object would allow its salary to reflect the organization's budget and the employee's particular contribution to the organization's performance.

By individuals contributing to a more accurate model of themselves and a more accurate characterization of how the systems they use should behave, then the society will be easier to understand, more efficient, more productive, and more satisfying, and its principles will be better adhered to.

As societies attempt to coordinate and control their members use of utilities and resources, individuals should have a means to influence the coordination and control based on their preferences. These are societal services that are beyond personal services. Decisions can be made centrally or collectively-when a system is dynamic so that decisions must be made in real-time, individuals would benefit from active systems that could intercede on their behalf. Examples of such institutions are banking, distributing electricity, determining routes for new roads, controlling traffic, managing telecommunication networks, manufacturing supply chains, and designing buildings or cities.

## 6.2 Designing and Engineering Agents

Software engineering principles applied to multiagent systems have yielded few new modeling techniques, despite many notable efforts. A comprehensive review of agent-oriented methodologies is contained in [15]. Many, such as Agent UML [24] and MAS-CommonKADS [14], are extensions of previous software engineering design processes. Others, such as Gaia [42], were developed specifically for agent modeling. These three have been investigated and applied the most. Other methodologies include the AAIL methodology [20], MaSE [7], Tropos [26], Prometheus [25], and ROADMAP [18]. Because agents are useful in such a broad range of applications, software engineering methodologies for multiagent systems should be a combination of efforts. A combination of principles and techniques will generally give a more flexible approach to fit a design team's particular expectations and requirements.

Multiagent systems can form the fundamental building blocks for software systems, even if the software systems do not themselves require any agent-like behaviors [16]. When a conventional software system is constructed with agents as its modules, it can exhibit the following characteristics and benefits [4, 10]:

- Agent-based modules, because they are active, more closely represent real-world things, which are the subjects of many applications. The modules can hold beliefs about the world, especially about themselves and others; if their behavior is consistent with their beliefs, then their behavior will be more predictable and reliable
- The modules can volunteer to be part of a software system, and can benevolently compensate for the limitations of other modules.
- Systems can be constructed dynamically, where the modules in a system and their interactions can be unknown until runtime. Because such modules can be added to a system one-at-a-time, software can continue to be customized over its lifetime, even by end-users as we are proposing to investigate here
- Because agents can represent multiple viewpoints and can use different decision procedures, they can produce more robust systems. The essence of multiple viewpoints and multiple decision procedures is redundancy, which is the basis for error detection and correction.

## 6.3 Bugs, Errors, and Redundancy

Software problems are typically characterized in terms of bugs and errors, which may be either transient or omnipresent. The general approaches for dealing with them are: (1) prediction and estimation, (2) prevention, (3) discovery, (4) repair, and (5) tolerance or exploitation. Bug estimation uses statistical techniques to predict how many flaws might be in a system and how severe their effects might be. Bug prevention is dependent on good software engineering techniques and processes. Good development and run-time tools can aid in bug discovery, whereas repair and tolerance depend on redundancy.

Indeed, redundancy is the basis for most forms of robustness. It can be provided by replication of hardware, software, and information, and by repetition

of communication messages. Redundant code cannot be added arbitrarily to a software system, just as steel cannot be added arbitrarily to a bridge. A bridge is made stronger by adding beams that are not identical to ones already there, but that have equivalent functionality. This turns out to be the basis for robustness in software systems as well: there must be software components with equivalent functionality, so that if one fails to perform properly, another can provide what is needed. The challenge is to design the software system so that it can accommodate the additional components and take advantage of their redundant functionality.

We hypothesize that agents are a convenient level of granularity at which to add redundancy and that the software environment that takes advantage of them is akin to a society of such agents, where there can be multiple agents filling each societal role Error! Reference source not found.. Agents by design know how to deal with other agents, so they can accommodate additional or alternative agents naturally.

Fundamentally, the amount of redundancy required is well specified by information theory. If we want a system to provide  $n$  functions robustly, we must introduce  $m \times n$  agents, so that there will be  $m$  ways of producing each function. Each group of  $m$  agents must understand how to detect and correct inconsistencies in each other's behavior, without a fixed leader or centralized controller. If we consider an agent's behavior to be either correct or incorrect (binary), then, based on a notion of Hamming distance for error-correcting codes,  $4m$  agents can detect  $m - 1$  errors in their behavior and can correct  $(m - 1)/2$  errors.

Redundancy must also be balanced with complexity, which is determined by the number and size of the components chosen for building a system. That is, adding more components increases redundancy, but also increases the complexity of the system.

An agent-based system can cope with a growing application domain by increasing the number of agents, each agent's capability, or the computational and infrastructure resources that make the agents more productive. That is, either the agents or their interactions can be enhanced, but to maintain the same redundancy  $n$ , they would have to be enhanced by a factor of  $n$ .

N-version programming, also called dissimilar software, is a technique for achieving robustness first considered in the 1970's. It consists of  $N$  separately developed implementations of the same functionality. Although it has been used to produce several robust systems, it has had limited applicability, because (1)  $N$  independent implementations have  $N$  times the cost, (2)  $N$  implementations based on the same flawed specification might still result in a flawed system, and (3) each change to the specification will have to be made in all  $N$  implementations.

Database systems have exploited the idea of transactions: an atomic processing unit that moves a database from one consistent state to another. Consistent transactions are achievable for databases because the types of processing done are very regular and limited. Applying this to software execution requires that

the state of a software system be saved periodically (a checkpoint) so the system can return to that state if an error occurs.

#### 6.4 Consensus Ontologies: Modeling Objects, Resources, and Agents

A key to enabling agents to interact productively is for them to construct and maintain models of each other, as well as the passive components in their environment. Unfortunately, the agents' models will be mutually incompatible in syntax and semantics, not only due to the different things being modeled, but also due to mismatches in underlying hardware and operating systems, in data structures, and in usage. In attempting to model some portion of the real world, information models necessarily introduce simplifications and inaccuracies that result in semantic incompatibilities. However, the models must be related to each other and their incompatibilities resolved.

If there are  $n$  entities in the environment, then each would need a model of each of the other entities, resulting in  $n(n-1)/2$  models that must be maintained. This is infeasible for large domains. We solve this via two means. First, we propose that agents maintain and advertise models of themselves, resulting in a total of  $n$  models. For the second, we consider the source of the models. How should one agent represent another, and how should it acquire the information it needs to construct a model in that representation?

This has, we believe, a simple and elegant answer: the agent should presume that unknown agents are like itself, and it should choose to represent them as it does itself. Thus, as an agent learns more about other agents, it only has to encode any differences that it discovers. The resultant representation can be concise and efficient, and has the following advantages:

- An agent has to manage only one kind of model and one kind of representation.
- The same inference mechanisms that are used to reason about its own behavior can reason about the behaviors of other agents; an agent trying to predict what another will do has only to imagine what it itself would do in a similar situation.

The main criteria used by an agent to decide if it can contribute to a new problem are the relative importance of each feature (or dimension) of the problem, the degree of similarity with the agent's capabilities, and an estimate of the agent's capabilities relative to other agents' capabilities.

Each agent maximizes its problem-solving utility, and, thus, is rational. We portray an agent as a rational decision-maker that perceives and interacts with its environment. In our case, percepts and interactions are messages received from and sent to other agents. Agents are rational in the context of all other agents in their organization or institution, because they are aware of the other agents' constraints, preferences, intentions, and commitments and act accordingly.

However, such organizational or institutional knowledge typically comes from many independent sources, each with its own semantics. How can the information

from large numbers of such sources can be associated, organized, and merged? Our hypothesis is that a multiplicity of ontology fragments, representing the semantics of the independent sources, can be related to each other automatically without the use of a global ontology. That is, any pair of ontologies can be related indirectly through a semantic bridge consisting of many other previously unrelated ontologies, even when there is no way to determine a direct relationship between them. The relationships among the ontology fragments indicate the relationships among the sources, enabling the source information to be categorized and organized. Our investigation of the hypothesis has been conducted by relating numerous small, independently developed ontologies for several domains. A nice feature of our approach is that common parts of the ontologies reinforce each other, while unique parts are deemphasized. The result is a consensus ontology.

The problem we are addressing is familiar and many solutions have been proposed, ranging from requiring search criteria to be more precise, to constructing more intelligent search engines, or to requiring sources to be more precise in describing their contents. A common theme for all of the approaches is the use of ontologies for describing both requirements and sources [34, 43]. Unfortunately, ontologies are not a panacea unless everyone adheres to the same one, and no one has yet constructed an ontology that is comprehensive enough (in spite of determined attempts to create one [33, 1], such as the Cyc Project [21], underway since 1984). Moreover, even if one did exist, it probably would not be adhered to, considering the dynamic and eclectic nature of the Web and other information sources.

There are three approaches for relating information from large numbers of independently managed sites: (1) all sites will use the same terminology with agreed-upon semantics (improbable), (2) each site will use its own terminology, but provide translations to a global ontology (difficult, and thus unlikely), and (3) each site will have a small, local ontology that will be related to those from other sites (described herein). We hypothesize that the small ontologies can be related to each other automatically without the use of a global ontology. That is, any pair of ontologies can be related indirectly through a semantic bridge consisting of many other previously unrelated ontologies, even when there is no way to determine a direct relationship between them. Our methodology relies on sites that have been annotated with ontologies [27], which is consistent with several visions for the Semantic Web [2]. The domains of the sites must be similar—else there would be no interesting relationships among them—but they will undoubtedly have dissimilar ontologies, because they will have been annotated independently.

Some researchers have attempted to merge a pair of ontologies in isolation, or merge a domain-specific ontology into a global, more general ontology [38]. To our knowledge, no one has previously tried to reconcile a large number of closely related, domain-specific ontologies. We have evaluated our methodology by applying it to a large number of independently constructed ontologies.

When two agents communicate, they must reconcile their semantics. This will be seemingly impossible if their ontologies share no concepts. However, if

their ontologies share concepts with a third ontology, then the third ontology might provide a semantic bridge to relate all three. Note that the agents do not have to relate their entire ontologies, only the portions needed to respond to the request.

The difficulty in establishing a bridge will depend on the semantic distance between the concepts, and on the number of ontologies that comprise the bridge. Our methodology is appropriate when there are large numbers of small ontologies—the situation we expect to occur in large and complex information environments. Our metaphor is that a small ontology is like a piece of a jigsaw puzzle. It is difficult to relate two random pieces of a jigsaw puzzle until they are constrained by other puzzle pieces. We expect the same to be true for ontologies.

In attempting to relate two ontologies, a system might be unable to find correspondences between concepts because of insufficient constraints and similarity among their terms. However, trying to find correspondences with other ontologies might yield enough constraints to relate the original two ontologies. As more ontologies are related, there will be more constraints among the terms of any pair, which is an advantage. It is also a disadvantage in that some of the constraints might be in conflict. We make use of the preponderance of evidence to resolve these statistically.

We conducted experiments in three domains as follows: We asked one group of 53 graduate students in computer science to construct a small ontology for the Humans/People/Persons domain, a second group of 28 students to construct a small ontology for the Buildings domain, and finally a third group of 26 students to construct a small ontology for the Sports domain. The ontologies were written in OWL and were required to contain at least 8 classes with at least 4 levels of subclasses.

We merged the files in each of the three domains. In the Humans/People/Persons ontology domain the component ontologies described 864 classes, while the merged ontology contained 281 classes in a single graph with a root node of the OWL concept owl:Thing. We constructed a consensus ontology by first counting the number of times classes and subclass links appeared in the component ontologies when we performed the merging operation. For example, the class Person and its matching classes appeared 14 times. The subclass link from Mammals (and its matches) to Humans (and its matches) appeared 9 times. We termed these numbers the "reinforcement" of a concept. After removing the nodes and links that were not reinforced, the resultant consensus ontology contained 36 classes related by 62 subclass links.

A consensus ontology is perhaps the most useful for information retrieval by humans, because it represents the way most people view the world and its information. For example, if most people wrongly believe that crocodiles are a kind of mammal, then most people would find it easier to locate information about crocodiles if it were placed in a mammals grouping, rather than where it factually belonged.

## 6.5 Agent-Based Web Services

Typical agent architectures have many of the same features as Web services. Agent architectures provide yellow-page and white-page directories, where agents advertise their distinct functionalities and where other agents search to locate the agents in order to request those functionalities. However, agents extend Web services in several important ways:

- A Web service knows only about itself, but not about its users/clients/customers. Agents are often self-aware, and gain awareness of the capabilities of other agents as interactions among the agents occur. This is important, because without such awareness a Web service would be unable to take advantage of new capabilities in its environment, and could not customize its service to a client, such as by providing improved services to repeat customers.
- Web services, unlike agents, are not designed to use and reconcile ontologies. If a client and provider of a service have different semantics, the result of invoking the service would be incomprehensible.
- Agents are inherently communicative, whereas Web services are passive until invoked. Agents can provide alerts and updates when new information becomes available. Current standards and protocols make no provision for even subscribing to a service to receive periodic updates.
- A Web service, as currently defined and used, is not autonomous. Autonomy is a characteristic of agents, and also a characteristic of many envisioned Internet-based applications. Autonomy is in natural tension with coordination or with the higher-level notion of a commitment. To be coordinated with other agents or to keep its commitments, an agent must relinquish some of its autonomy. It would attempt to coordinate with others where appropriate and keep its commitments, but would exercise its autonomy in agreeing to those commitments in the first place.
- Agents are cooperative, and by forming coalitions can provide higher-level and more comprehensive services. Current standards for Web services are just beginning to address composition.

Suppose an application needs simply to sort some data items, and suppose there are five Web sites that offer sorting services described by their input data types, output data type, time complexity, space complexity, and quality: one is faster, one handles more data types, one is often busy, one returns a stream of results, while another returns a batch, and one costs less. An application could take one of the following possible approaches:

- The application invokes services randomly until one succeeds
- The application ranks services and invokes them in order until one succeeds
- The application invokes all services and reconciles the results
- The application contracts with one service after requesting bids
- Services self-organize into a team of sorting services and route requests to the best one.

The last two require that the services behave like agents. Furthermore, the last two are scalable and robust, because they take advantage of the redundancy that is available.

## 7 Conclusions

Applications are beginning to span enterprises with autonomous components. Moreover, the Web is making it possible for individuals to make available not only information, but also behavior. Work on the Semantic Web is making it possible for mutual understanding to be reached among autonomous components. The ongoing research on consensus software enables individuals to participate more directly in the behavior of their governing institutions.

## References

1. E. Agirre, O. Ansa, E. Hovy, and D. Martinez, "Enriching very large ontologies using the WWW," in *Proceedings of the Ontology Learning Workshop*, ECAI, Berlin, Germany, July 2000.
2. Tim Berners-Lee, James Hendler, and Ora Lassila, "The Semantic Web," *Scientific American*, Vol. 284, No. 5, May 2001, pp. 34–43.
3. Antoine Beugnard, Jean-Marc Jezequel, Noel Plouzeau, and Damien Watkins, "Making Components Contract Aware," *IEEE Computer*, Vol. 32, No. 7, July 1999, pp. 38–45.
4. Helder Coelho, Luis Antunes, and Luis Moniz, "On Agent Design Rationale," in *Proceedings of the XI Simposio Brasileiro de Inteligencia Artificial (SBIA)*, Fortaleza (Brasil), October 17–21, 1994, pp. 43–58.
5. Philip R. Cohen and Hector J. Levesque, "Persistence, Intention, and Commitment," in *Intentions in Communication*, Philip R. Cohen, Jerry Morgan, and Martha E. Pollack, eds., MIT Press, 1990.
6. J. R. Commons, *Institutional Economics: Its Place in Political Economy*, University of Wisconsin Press, Madison, WI, 1934.
7. Scott DeLoach, "Analysis and Design using MaSe and agentTool," *Proc. 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, 2001.
8. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
9. Les Gasser, "Social conceptions of knowledge and action: DAI foundations and open systems semantics," *Artificial Intelligence*, Vol. 47, 1991, pp. 107–138.
10. Michael N. Huhns, "Interaction-Oriented Programming," *Agent-Oriented Software Engineering*, Paulo Ciancarini and Michael Wooldridge, editors, Springer Verlag, Lecture Notes in AI, Volume 1957, Berlin, 2001, pp. 29–44.
11. Michael N. Huhns, "Agent Teams: Building and Implementing Software," *IEEE Internet Computing*, Vol. 4, No. 1, January/February 2000, pp. 90–92.
12. Michael N. Huhns, "Multiagent-Oriented Programming," *Intelligent Agents and Their Potential for Future Design and Synthesis Environments*, Ahmed K. Noor and John B. Malone, editors, NASA Langley Research Center, Hampton, VA, February 1999, pp. 215–238.
13. Michael N. Huhns and Munindar P. Singh, "A Multiagent Treatment of Agenthood," *Applied Artificial Intelligence: An International Journal*, Vol. 13, No. 1–2, January–March 1999, pp. 3–10.
14. C.A. Iglesias, M. Garijo, J. C. Gonzales, and R. Velasco, "Analysis and Design of Multi-Agent Systems using MAS-CommonKADS," *Proc. AAAI'97 Workshop on agent Theories, Architectures and Languages*, Providence, USA, 1997.

15. C. Iglesias, M. Garijo, and J. Gonzalez, "A survey of agent-oriented methodologies," in J. Muller, M. P. Singh, and A. S. Rao, editors, *Proc. 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98)*, Springer-Verlag, Heidelberg, Germany, 1999.
16. Nicholas R. Jennings, "On Agent-Based Software Engineering," *Artificial Intelligence*, Vol. 117, No. 2, 2000, pp. 277–296.
17. Nicholas R. Jennings, "Commitments and conventions: The foundation of coordination in multi-agent systems," *The Knowledge Engineering Review*, Vol. 2, No. 3, 1993, pp. 223–250.
18. T. Juan, A. Pearce, and L. Sterling, "Extending the Gaia Methodology for Complex Open Systems," *Proceedings of the 2002 Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 2002.
19. Elizabeth A. Kendall, Margaret T. Malkoun, and Chong Jiang, "Multiagent System Design Based on Object-Oriented Patterns," *Journal of Object-Oriented Programming*, June 1997, pp. 41–47.
20. David Kinny and Michael Georgeff, "Modelling and Design of Multi-Agent Systems," in J.P. Muller, M.J. Wooldridge, and N.R. Jennings, eds., *Intelligent Agents III — Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, Berlin, 1997, pp. 1–20.
21. , D. B. Lenat and R. V. Guha, *Building Large Knowledge-Based Systems*, Addison-Wesley, Reading, MA, 1990.
22. David L. Martin, Adam J. Cheyer, and Douglas B. Moran, "The Open Agent Architecture: A framework for building distributed software systems," *Applied Artificial Intelligence*, Vol. 13, No. 1–2, 1999, pp. 92–128.
23. D. C. North, *Institutions, Institutional Change, and Economic Performance*, Cambridge University Press, Cambridge, MA, 1990.
24. James Odell, H. Van Dyke Parunak, and Bernhard Bauer, "Extending UML for Agents," *Proceedings of the Agent-Oriented Information Systems Workshop*, Gerd Wagner, Yves Lesperance, and Eric Yu, editors, Austin, TX, 2000.
25. Lin Padgham and M. Winikoff, "Prometheus: A Methodology for Developing Intelligent Agents," *Proc. Third International Workshop on Agent-Oriented Software Engineering*, July, 2002, Bologna, Italy.
26. A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos, "A knowledge level software engineering methodology for agent oriented programming," *Proceedings of Autonomous Agents*, Montreal CA, 2001.
27. J. M. Pierre, "Practical Issues for Automated Categorization of Web Sites," *Electronic Proc. ECDL 2000 Workshop on the Semantic Web*, Lisbon, Portugal, 2000. <http://www.ics.forth.gr/proj/isst/SemWeb/program.html>
28. M. J. Pont and E. Moreale, "Towards a Practical Methodology for Agent-Oriented Software Engineering with C++ and Java," Leicester University Technical Report 96-33, December 1996.
29. Anand S. Rao and Michael P. Georgeff, "Modeling rational agents within a BDI-architecture," in *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 1991, pp. 473–484.
30. Yoav Shoham, "Agent-Oriented Programming," *Artificial Intelligence*, Vol. 60, No. 2, June 1993, pp. 51–92.
31. Charles Simonyi, "The Future is Intentional," *IEEE Computer*, Vol. 32, No. 5, May 1999, pp. 56–57.
32. Munindar P. Singh and Michael N. Huhns, "Social Abstractions for Information Agents," in *Intelligent Information Agents*, Matthias Klusch, ed., Kluwer Academic Publishers, Boston, MA, 1999.

33. K. Stoffel, M. Taylor, and J. Hendler, "Efficient Management of Very Large Ontologies," in *Proceedings of American Association for Artificial Intelligence Conference (AAAI-97)*, AAAI/MIT Press, pp. 442-447, 1997.
34. W. Swartout and A. Tate, "Ontologies," *IEEE Intelligent Systems*, vol. 14, no. 1, pp. 18-19, 1999.
35. Clement Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley Longman, 1998.
36. Jose M. Vidal, Paul A. Buhler, and Michael N. Huhns, "Inside an Agent," *IEEE Internet Computing*, Vol. 5, No. 1, January/February 2001, pp. 86-90.
37. Peter Wegner, "Why Interaction is More Powerful Than Algorithms," *Communications of the ACM*, Vol. 40, No. 5, May 1997, pp. 80-91.
38. Gio Wiederhold, "An Algebra for Ontology Composition," in *Proc. Monterey Workshop on Formal Methods*, U.S. Naval Postgraduate School, pp. 56-61, 1994.
39. Darrell Woelk, Michael Huhns, and Christine Tomlinson, "Uncovering the Next Generation of Active Objects," *Object Magazine*, July-August 1995, pp. 33-40.
40. Michael J. Wooldridge, "Agent-Based Software Engineering," *IEE Proceedings on Software Engineering*, Vol. 144, No. 1, February 1997, pp. 26-37.
41. Michael J. Wooldridge and Nicholas R. Jennings, "Software Engineering with Agents: Pitfalls and Pratfalls," *IEEE Internet Computing*, Vol. 3, No. 3, May/June 1999, pp. 20-27.
42. Michael Wooldridge, N. R. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," *Journal of Autonomous Agents and Multi-Agent Systems*, 2000.
43. K.T. Yao, I.Y. Ko, R. Eleish, and R. Neches, "Asynchronous Information Space Analysis Architecture Using Content and Structure Based Service Brokering," in *Proceedings of Fifth ACM Conference on Digital Libraries (DL 2000)*, San Antonio, Texas, June 2000.
44. Edward Yourdon, "Java, the Web, and Software Development," *IEEE Computer*, August 1996, pp. 25-30.