Consensus Software: Robustness and Social Good via Widespread Multiagent Development

Michael N. Huhns
University of South Carolina
Department of Computer Science and Engineering
Columbia, SC 29208 USA
+1-803-777-5921 phone; +1-803-777-3767 fax
huhns@sc.edu

ABSTRACT

This paper describes two complementary research thrusts: (1) it investigates how software robustness can be achieved pragmatically through the use of large-scale multiagent systems, and (2) it investigates how a broader spectrum of people can contribute to the production and customization of software, also through the use of multiagent systems. The paper first hypothesizes 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. We describe our initial experiments in verifying our hypothesis and present results that 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 software 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. The result enables individuals to participate more directly in the behavior of their governing institutions.·

KEYWORDS

Robust software; multiagent systems; service-oriented computing; ontologies; consensus software

· The US National Science Foundation supported this work under grant number IIS-0083362.

1. INTRODUCTION

Computer systems are now entrusted with control of global telecommunications, electric power distribution, water supplies, airline traffic, weapon systems, and the manufacturing and distribution of goods. Such tasks typically are complex, involve massive amounts of data, affect numerous connected devices, and are subject to the uncertainties of open environments like the Internet. Our society has come to expect uninterrupted service from these systems. Unfortunately, when problems arise, humans are unable to cope with the complexity of the systems and the speed with which they must be repaired. Increasingly, the result is that critical missions are in jeopardy.

To cope with this situation, researchers are investigating self-healing systems, which detect problems autonomously and continue operating by fixing or bypassing the malfunction [33]. The techniques employed include redundant hardware, error-correction codes, and models of how a system *should* behave, so that the system can recognize when it *mis*behaves.

The most common technique for robustness in hardware—redundant components—is inappropriate for software, because having identical copies of a module provides no additional benefit. For software, reliability is thus a more difficult and still unresolved problem [3, 4]. The amount of money lost due just to software errors is conservatively estimated at US$40B annually. To produce higher quality software, researchers are trying to define more principled methodologies for software engineering [10]. They are also looking to new technologies, such as multiagent systems, and this leads to an interest in combining software engineering methodologies with multiagent systems [30].

There are at least three ways that software engineering intersects multiagent systems:

- Multiagent systems can be used to aid traditional software engineering, such as by agent-based or agent-supplemented CASE tools
- Traditional or new software engineering techniques can be used to build multiagent systems; e.g., UML has proven to be useful for conventional software, so Agent-Based UML and similar efforts are underway to extend it to support the development of agents [31]
- Conventional software can be constructed out of agents, and software engineering can be used in this endeavor.

The focus of this paper is on the last, but extended to widespread development by people other than expert developers. The behavior of the resultant systems will depend on their construction and the environment in which they operate. When the system contains a number of components that interact with each other and a complex environment, the behavior can be difficult to predict and control. Traditional software interfaces are rigid. Often the slightest error in the implementation of a component can have far-reaching repercussions on the behavior of the entire system. However, the output of a component may be erroneous because of its malfunctioning, its environment being out of its design range, or an erroneous input from another component.

We are pursuing five research threads that are converging to enable a new class of reliable software systems that meet the expectations of large groups of users or even the general public. 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 [45]. 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.

Third, 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.

Fourth, 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.

Fifth, 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 (workflow) 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.

2. 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 a `Person`.) 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 [29]. 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 an economic downturn, an organization facing a decline in its revenues and budget might choose a common, centralized response: 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 [9]. 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 Common'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 individuals' 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, managing telecommunication networks, manufacturing supply chains, and designing buildings or cities.

As a specific example, the behavior of the traffic lights at an intersection should be determined by the wishes of the vehicles passing through the intersection at any given time, not by a traffic engineer's prior estimate of those wishes.

3. AGENT-ORIENTED SOFTWARE SYSTEM DEVELOPMENT

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 [21]. Many, such as Agent UML [31] and MAS-CommonKADS [20], are extensions of previous software engineering design processes. Others, such as Gaia [43], were developed specifically for agent modeling. These three have been investigated and applied the most. Other methodologies include the AAII methodology [26], MaSE [11], Tropos [34], Prometheus [32], and ROADMAP [23]. 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 expectations and requirements.

Multiagent systems can form the fundamental building blocks for software systems, even if the software systems do not themselves require agent-like behaviors [22]. When a conventional software system is constructed with agents as its modules, it can exhibit the following characteristics and benefits [8, 18]:

- 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.

4. 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. 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, $4 \times m$ 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.

5. ARCHITECTURE AND PROCESS

Suppose there are a number of algorithms, each with strengths, weaknesses, and possibly errors. How can the algorithms be combined so that the strengths are exploited and the weaknesses or flaws are compensated or covered?

Three general approaches are evident in Figure 1. First, a preprocessor could choose the best algorithms to perform the task, based on known characteristics of each algorithm. Second, a postprocessor could choose the best result out of several executing algorithms. Third, the algorithms could decide as a group which ones should perform the task.

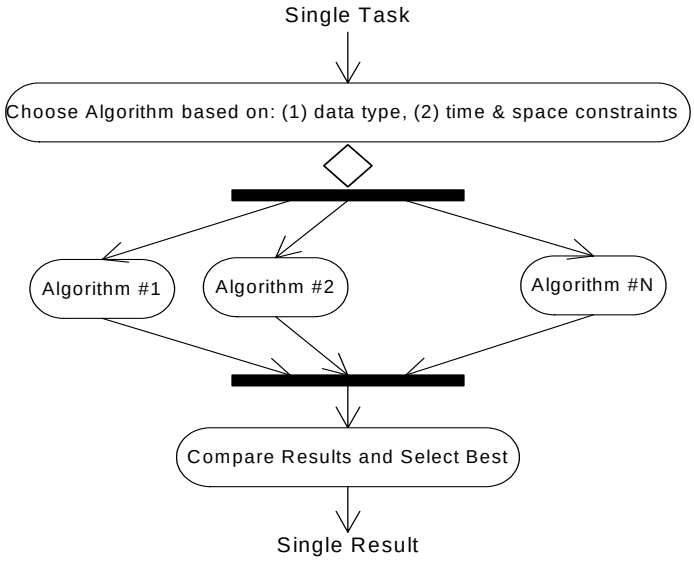

Figure 1. General architecture for combining *N* versions of an algorithm into a single, more robust system

The difficulties with the first two approaches are (1) the preprocessor might be flawed, (2) it is difficult to maintain the preprocessor as algorithms are added or changed, and (3) the postprocessor wastes resources, because several algorithms work on the data and their results have to be compared.

The third approach requires distributed decision-making, which is not an ability of conventional algorithms, so we investigated what generic ability could be added to an algorithm to enable it to participate in a distributed decision. The generic capability has the characteristics of an agent, and distributing the centralized functions into the different modules creates a multiagent system. Each agent would have to know its role as well as (1) something about its own algorithm, such as its time and space complexity, and input and output data structures; (2) the complexity and reliability of other agents; and (3) how to communicate, negotiate, compare results, and manage reputations and trust.

6. 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 [38].

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 [40, 44]. 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 [1, 39], such as the Cyc Project [27], 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 [35], which is consistent with several visions for the Semantic Web [5, 6, 13]. 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 [41]. 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.

7. AGENT-BASED WEB SERVICES

The World-Wide Web was designed for humans. It is based on a simple concept: information consists of pages of text and graphics that contain links, and each link leads to another page of information, with all of the pages meant to be viewed by a person. The constructs used to describe and encode a page, the Hypertext Markup Language, describe the appearance of the page, but not its contents. Software agents don't care about appearance, but rather the contents. The Semantic Web [5] will make the Web more accessible to agents by making use of semantic constructs, such as ontologies represented in OWL, RDF, and XML, so that agents can *understand* what is on a page. It will also incorporate functionality in the form of Web services.

Our architecture is compatible with the standards being formulated for the Semantic Web and its Web services and, in fact, relies on them to provide the infrastructure needed for installation, deployment, and activation of consensus software. By themselves, Web services are inadequate for specifications of personalized behavior, because they are passive until discovered and invoked. People need components that will proactively take action on their behalf.

The architecture for Web services is founded on principles and standards for connection, communication, description, and discovery. For providers and requestors of services to be connected and exchange information, there must be a common language. This is provided by the Extensible Markup Language (XML). A common protocol is required for systems to communicate with each other, so that they can request services, such as to schedule appointments, order parts, and deliver information. This is provided by the Simple Object Access Protocol (SOAP). The services must be described in a machine-readable form, where the names of functions, their required parameters, and their results can be specified. This is provided by the Web Services Description Language (WSDL). Finally, clients—users and businesses—need a way to find the services they need. This is provided by Universal Description, Discovery, and Integration (UDDI), which specifies a registry or "yellow pages" of services.

Besides standards for XML, SOAP, WSDL, and UDDI, there is a need for broad agreement on the semantics of specific domains. This is provided by the Resource Description Framework (RDF) and ontologies expressed in OWL. Web services currently involve a single client accessing a single server, but soon applications will demand federated servers with multiple clients sharing results. Cooperative peer-to-peer solutions will have to be managed, and this is an area where agents have excelled. In doing so, agents can balance cooperation with

the interests of their owner. Composing Web services requires capturing patterns of semantic and pragmatic constraints on how services may participate in different compositions. It also requires tools to help reject unsuitable compositions so that only acceptable systems are built. The following challenges have not yet been met by current implementations and standards for Web services:

Information Semantics. The composer and the member services must agree on the semantics of the information that they exchange.

Collaboration. To perform even simple protocols reliably, service providers must ensure that the parties to an interaction agree on its current state and where they desire to take it. This requires elements of teamwork through (1) persistence of the computations, (2) ability to manage context, and (3) retrying.

Personalization. Effective usage of services often requires customization of the compositions in a manner that is context sensitive, especially with respect to user needs. This requires (1) learning a customer's preferences, (2) mixed-initiative interactions, offering guidance to a customer, and (3) acting on behalf of a user, which is limited to ensure that a user's autonomy is not violated.

Exception Conditions. To construct virtual enterprises dynamically to provide more appropriate goods and services to common customers requires the ability to (1) construct teams, (2) enter into multiparty deals, (3) handle authorizations and commitments, and (4) accommodate exceptions.

Service Location. Recommendations must be provided to help customers find relevant, high quality, and trustworthy services. This requires a means to find, obtain, and aggregate evaluations.

Distributed Decision-Making. Decision-making will be distributed across the composed services, which requires intelligent decisions by each service so the composed services can collaborate and compete appropriately, while accommodating exceptions.

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 date 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.

8. EXPERIMENTAL RESULTS FOR ROBUSTNESS

We collected a number of algorithms in four domains—geographic location control, sorting, list-reversing, and evaluation of postfix arithmetic expressions—each written by a different person and therefore having different input and output signatures and performance characteristics. The programmers were undergraduate computer science majors and the work was done as standard homework assignments. The students were unaware that their algorithms would be used in our tests for robustness, so the algorithms did not have any special features that would bias our results. We converted each algorithm into an agent composed of the algorithm without any modifications and a wrapper for that algorithm. The wrapper knows nothing about the inner workings of its associated algorithm. It has knowledge only about the external characteristics of its algorithm, such as the data type(s) it requires and produces, its time complexity, and its space complexity. The algorithms were written in Java and the wrappers in JADE.

The input-process-output systems begin by a notification sent to the Initiator agent about data to be processed, which notifies the processing agents. Upon receiving data, each agent determines whether or not it can process it successfully, based on the type of the data and its own knowledge of what types it can handle. If the agent believes it can succeed, it broadcasts an INFORM message to every other agent specifying its intention, along with a measure of its expected performance.

The decision of which agent to choose among those that are capable of processing the input data is made in a distributed manner: upon receiving INFORM messages from other agents, each agent compares its own performance against those in the messages. If the agent has the best performance, it will run its algorithm and send the results back to the system. If it does not have the best performance, it will do nothing. Also, once they receive the data from the system, the agents will wait for INFORM messages for only a limited amount of time; this avoids waiting infinitely long for messages from agents that either have problems sending a message or are unable to process the data.

In preliminary experiments, we asked 30 students to implement an agent for a control-system application as a concurrently executing Java thread and interacting through a base class environment. For controlling the location of objects forming a geometric circle in a plane, the agents each understand what a circle is, what it means to be part of a circle, where the nearest agents are located, and an estimate of how close the group is to being in a circle. The agents can reason about where they should be on a circle and the direction they should move to get there. They also have the ability to help move nearby agents that do not seem to be located or moving properly. Into this environment, we have introduced a few agents that do not have the ability to move properly or are stationary. The group overcomes this and produces an acceptable circle. We have anecdotal evidence, via one comparison, that such an implementation can be constructed more rapidly and robustly than a conventional C++ implementation.

We have also experimented with a postprocessing implementation of distributed decision-making. Even with the simplicity of the current implementations, results showing improvement in robustness due to redundancy were obtained. As a group, the agents process data better than any one of them alone.

We collected one set of 25 algorithms for reversing a doubly linked list and another set for sorting a list. Different people wrote each algorithm. For sorting, no specifications were given to the programmers, so the algorithms all have different data and performance characteristics. For list reversing, the class structure (i.e., method signatures) was specified, so the differences among the algorithms are in performance and correctness.

We converted each algorithm into an agent, composed of the algorithm written in JAVA and a wrapper written in JADE. The wrapper knows only about the signature of its algorithm, and nothing about its inner workings.

Our experiments verified that the same wrapper can be used for both domains. We also verified our hypothesis that more algorithms give better results than any one alone. Further, we investigated both a distributed preprocessor and a centralized postprocessor for combining the agents' functionality, and found that the postprocessor is generally better, but performs worse for large data sets or selected algorithms with long execution times.

The eventual outcome for software development is that developers will spend more time on algorithm development and less on debugging, because different algorithms will likely have errors in different places and can cover for each other.

9. References

[1] E. Agirre, O. Ansa, E. Hovy, and D. Martínez, "Enriching very large ontologies using the WWW," in *Proceedings of the Ontology Learning Workshop*, ECAI, Berlin, Germany, July 2000.

[2] T. Anderson and R. Kerr, "Recovery blocks in action: A system supporting high reliability," *Proc. 2nd International Conference on Software Engineering*, October 13-15, 1976, San Francisco, CA, p.447–457.

[3] Algirdas Avizienis, "Fault-Tolerant Systems," *IEEE Transactions on Computers*, Vol. 25, No. 12, 1976, pp. 1304-1312.

[4] Algirdas Avizienis, "Toward Systematic Design of Fault-Tolerant Systems," *IEEE Computer*, Vol. 30, No. 4, 1997, pp. 51-58.

[5] Tim Berners-Lee, James Hendler, and Ora Lassila, "The Semantic Web," *Scientific American*, Vol. 284, No. 5, May 2001, pp. 34-43.

[6] Tim Berners-Lee, *Weaving the Web*, Harper, San Francisco, CA, 1999.

[7] K.N. Chandy and C. V. Ramamoorthy, "Rollback and recovery strategies for computer programs," *IEEE Transactions on Computers*, June 1972, pp. 59-65.

[8] Helder Coelho, Luis Antunes, and Luis Moniz, "On Agent Design Rationale," in *Proceedings of the XI Simpósio Brasileiro de Inteligência Artificial (SBIA)*, Fortaleza (Brasil), October 17-21, 1994, pp. 43-58.

[9] J. R. Commons, *Institutional Economics: Its Place in Political Economy*, University of Wisconsin Press, Madison, WI, 1934.

[10] Brad J. Cox, "Planning the Software Industrial Revolution," *IEEE Software*, Nov. 1990, pp. 25-33.

[11] Scott DeLoach, "Analysis and Design using MaSe and agentTool," *Proc. 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, 2001.

[12] D.E. Eckhardt and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, 1985, pp. 1511-1517.

[13] J. Heflin and J. Hendler, "Dynamic Ontologies on the Web," in *Proc. 17th National Conference on AI (AAAI-2000)*, AAAI Press, Menlo Park, CA, pp. 443-449, July 2000.

[14] Michael N. Huhns, "Software Agents: The Future of Web Services," in *Software Agent Technology*, Hua Tianfiled, Jörg Müller, Ryszard Kowalcyk, and Rainer Unland, editors, Springer Verlag, Berlin, 2003.

[15] Michael N. Huhns and Vance T. Holderfield, "Robust Software," *IEEE Internet Computing*, vol. 6, no. 2, pp. 78-80, March/April 2002.

[16] Michael N. Huhns, "Agents as Web Services," *IEEE Internet Computing*, vol. 6, no. 4, July/August 2002, pp. 93-95.

[17] Michael N. Huhns and Larry M. Stephens, "Semantic Bridging of Independent Enterprise Ontologies," (with Larry M. Stephens), in *Enterprise Inter- and Intra-Organizational Integration: Building International Consensus*, Kurt Kosanke et al., editors, Kluwer Academic Publishers, Boston, MA, 2002.

[18] 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.

[19] Michael N. Huhns, editor, *Distributed Artificial Intelligence*, Pitman/Morgan Kaufmann, London, 1987.

[20] 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.

[21] C. Iglesias, M. Garijo, and J. Gonzalez, "A survey of agent-oriented methodologies," 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.

[22] Nicholas R. Jennings, "On Agent-Based Software Engineering," *Artificial Intelligence*, Vol. 117, No. 2, 2000, pp. 277-296.

[23] 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.

[24] Jeffrey O. Kephart and David M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, Vol. 36, No. 1, January 2003, pp. 41-50.

[25] K.H. Kim and Howard O. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications," *IEEE Transactions on Computers*, Vol. 38, No. 5, May 1989, pp. 626-636.

[26] 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.

[27], D. B. Lenat and R. V. Guha, *Building Large Knowledge-Based Systems*, Addison-Wesley, Reading, MA, 1990.

[28] Bev Littlewood, Peter Popov, and Lorenzo Strigini, "Modelling software design diversity—a review," *ACM Computing Surveys*, Vol. 33, No. 2, June 2001, pp. 177-208.
[29] D. C. North, *Institutions, Institutional Change, and Economic Performance*, Cambridge University Press, Cambridge, MA, 1990.
[30] Hyacinth S. Nwana and Michael Wooldridge, "Software Agent Technologies," *BT Technology Journal*, Vol. 14, No. 4, 1996, pp. 68-78.
[31] 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.
[32] 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.
[33] Linda Dailey Paulson, "Computer System, Heal Thyself" *IEEE Computer*, Vol. 35, No. 8, August 2002, pp. 20-22.
[34] 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.
[35] 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
[36] Brian Randell and Jie Xu, "The Evolution of the Recovery Block Concept," *Software Fault Tolerance*, M. Lyu, ed., Trends in Software, J. Wiley, 1994, pp.1-22.
[37] John R. Rose, Michael N. Huhns, Soumik Sinha Roy, and William H. Turkett, Jr. "An Agent Architecture for Long-Term Robustness," in *Proceedings First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, Bologna, Italy, July 15-19, 2002.
[38] Amit P. Sheth and James A. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, Vol. 22, no. 3, pp. 183-236, 1990.
[39] 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.
[40] W. Swartout and A. Tate, "Ontologies," *IEEE Intelligent Systems*, vol. 14, no. 1, pp. 18-19, 1999.
[41] Gio Wiederhold, "An Algebra for Ontology Composition," in *Proc. Monterey Workshop on Formal Methods*, U.S. Naval Postgraduate School, pp. 56-61, 1994.
[42] 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.
[43] 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.
[44] 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.
[45] Edward Yourdon, "Java, the Web, and Software Development," *IEEE Computer*, August 1996, pp. 25-30.