Achieving Software Robustness via Large-Scale Multiagent Systems

Michael N. Huhns, Vance T. Holderfield, and Rosa Laura Zavala Gutierrez

University of South Carolina, Department of Computer Science and Engineering,
Columbia, SC 29208, USA
{Huhns, Holderfield, Zavalagu}@engr.sc.edu
http://www.cse.sc.edu/∼huhns

Abstract. This paper describes how multiagent systems can be used to achieve robust software, one of the major goals of software engineering. The paper first positions itself within the software engineering domain. It then develops the hypothesis 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 conclude by speculating on the implications of multiagent-based redundancy for software development.

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 are typically 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, companies and researchers are investigating self-monitoring and self-healing systems, which detect problems autonomously and continue operating by fixing or bypassing the malfunction [25]. The techniques employed include redundant hardware, error-correction codes, and, most importantly, models of how a system *should* behave, so that the system can recognize when it *mis*behaves.

The most common technique for hardware, redundant components, is inappropriate for software, because having identical copies of a module provides no benefit. Software reliability is thus a more difficult and still unresolved problem [2, 3]. 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 [6]. They are also looking to new technologies, such as multiagent systems, and this leads to a natural interest in combining the latest software engineering methodologies with multiagent systems [22].

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 [23]
- 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.

2 Background

2.1 Methodologies for MAS Modeling

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 Iglesias, et al. [13]. Many, such as Agent UML [23] and MAS-CommonKADS [12], are extensions of previous software engineering design processes. Others, such as Gaia [28], were developed specifically for agent modeling. These three have been investigated and applied the most. Other methodologies include the AAII methodology [18], MaSE[7], Tropos [26], Prometheus [24], and ROADMAP [16]. 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.

Agent UML (AUML) extends UML, which emphasizes things, relationships between things, and diagrams for grouping things and their relationships, by including agent interaction protocols. An extension to CommonKADS [12] includes notations from several object-oriented modeling techniques (pre-UML), and introduces seven knowledge models to take advantage of agent-oriented design. Gaia was formulated from an organization theory perspective. Its methodology is based on a system model consisting of roles, permissions, responsibilities, protocols, activities, liveness properties, and safety properties.

2.2 Benefits of an Agent-Oriented Approach

Multiagent systems can form the fundamental building blocks for software systems, even if the software systems do not themselves require any agent-like be-

haviors [15]. When a conventional software system is constructed with agents as its modules, it can exhibit the following characteristics:

- Agent-based modules, because they are active, more closely represent real-world things, which are the subjects of many applications
- 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
- Modules can negotiate with each other, enter into social commitments to collaborate, and can change their mind about their results
- Modules can *volunteer* to be part of a software system.

The benefits of building software out of agents are [5, 11]

1. Agents enable dynamic composibility, where the components of a system can be unknown until runtime
2. Agents allow interaction abstractions, where interactions can be unknown until runtime
3. Because agents can be added to a system one-at-a-time, software can continue to be customized over its lifetime, even potentially by end-users
4. 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.

2.3 Bugs, Errors, and Redundancy

Hardware robustness is typically characterized in terms of faults and failures; equivalently, software robustness is typically characterized in terms of bugs and errors. Faults and bugs are flaws in a system, whereas errors and failures are the consequences of encountering the flaws during the operation or execution of the system. The flaws may be either transient or omnipresent. The general aspects of dealing with flaws are the same for both hardware and software: (1) predict their occurrence, (2) prevent their occurrence, (3) estimate their severity, (4) discover them, (5) repair or remove them, and (6) mitigate or exploit them.

Fault and bug estimation uses statistical techniques to predict how many flaws might be in a system and how severe their effects might be. For example, when Windows XP was released by Microsoft, it was estimated that it still contained 60,000 bugs, based on the rate at which its bugs were being discovered. Bug prevention is dependent on good software engineering techniques and processes. Good development and run-time tools can aid in bug discovery, bug repair is knowledge-intensive, and mitigation depends 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. For years, NASA has made its satellites more robust by duplicating critical subsystems. If a hardware subsystem fails, there is an identical replacement ready to begin operating. The space shuttle has quadruple

redundancy, and will not be launched without all copies functioning. However, software redundancy has to be provided in a different way. Identical software subsystems will fail in identical ways, so extra copies do not provide any benefit.

Moreover, code cannot be added arbitrarily to a software system, just as steel cannot be added arbitrarily to a bridge. Bridges are 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 the 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 [10]. Agents by design know how to deal with other agents, so they can accommodate additional or alternative agents naturally. They also typically are able to negotiate over and reconcile different viewpoints.

Fundamentally, the amount of redundancy required is well specified by information and coding theory. Assume each software module in a system can behave either correctly or incorrectly (the basis for unit testing as used by most software development organizations) and is independent of the other modules (so they do not suffer from the same faults). Then two modules with the same intended functionality are sufficient to detect an error in one of them, and three modules are sufficient to correct the incorrect behavior (by voting, or choosing the best two-out-of-three). This is how parity bits work in code words. Unlike parity bits, and unlike bricks and steel bridge beams, however, the software modules cannot be identical, or else they would not be able to correct each other's errors.

If we want a system to provide n functionalities robustly, we must introduce $m \times n$ agents, so that there will be m ways of producing each functionality. Each group of m agents must understand how to detect and correct inconsistencies in each other's behavior. 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.

Fundamentally, redundancy must 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. This is just another form of the common software engineering problem of choosing the proper size of the modules used to implement a system. Smaller modules are simpler, but their interactions are more complicated because there are more modules.

An agent-based system can cope with a growing application domain by increasing the number of agents, each agent's capability, the computational resources available to each agent, or the infrastructure services needed by the agents to make them more productive. That is, either the agents or their inter-

actions can be enhanced, but to maintain the same degree of redundancy n, they would have to be enhanced by a factor of n.

To underscore the importance being given to redundancy and robustness, several initiatives are underway around the world to investigate them. IBM has a major initiative to develop autonomic computing—"a systemic view of computing modeled after the self-regulating autonomic nervous system." Systems that can run themselves incorporate many biological characteristics, such as self-healing (redundancy), adaptability to changing environments (reconfigurability), identity (awareness of their own resources), and immunity (automatic defense against viruses). An autonomic computing system will adhere to self-healing, not by "cellular regrowth," but by making use of redundant elements to act as replenishment parts. By taking advantage of redundant services located around the world, a better range of services can be provided for customers in business transactions.

For example, IBM's Tivoli Risk Manager monitors a network's health, protects it against attack, and heals it in the event of attack. Among its autonomic features is a monitoring function referred to as the "heartbeat" that tracks so-called "keepalive" messages from third party security products and gives administrators an early warning about failures in their security infrastructure. If a connection is lost, the heartbeat monitor issues an alert to the Risk Manager to take action or notify a human operator.

Exemplifying extreme redundancy in hardware, HP Labs has built a massively parallel computer, the Teramac, with 220,000 known defects, but it still yields correct results. As long as there is sufficient communication bandwidth to find and use healthy resources, it can tolerate the defects. Allowing so many defects enables the computer to be built cheaply.

The National Science Foundation has launched the Infrastructure for Resilient Internet Systems (IRIS) project, which is a five-year initiative to produce a robust, decentralized, and secure Internet infrastructure. The infrastructure will be developed using distributed hash table (DHT) technology, which can prevent all the data in a network from becoming vulnerable if one server crashes. Rather than centralizing the data in a single server, each server contains a partial list of the data's storage location; the challenge lies in developing a lookup algorithm that can locate data using the fewest possible steps. IRIS grew out of rising worries of the Internet's susceptibility to failure and attacks from viruses, worms, and possibly cyberterrorists.

2.4 N-Version Programming

N-version programming [8, 20], also called dissimilar software and design diversity, is a technique for achieving robustness first considered in the 1970's. It consists of N disparate and 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, (3) the resultant system might have N

times the maintenance cost (e.g., each change to the specification will have to be made in all N implementations), and (4) the N versions must be combined without introducing additional errors. Our work addresses this last problem.

2.5 Transaction Checkpointing, Rollback, and Recovery

Database systems have exploited the idea of transactions for maintaining the consistency of their data. A transaction is an atomic unit of processing 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 idea to general software execution requires that the state of a software system be saved periodically (a checkpoint) so that the system can return to that state if an error occurs. The system then returns to that state and processes other transactions or alternative software modules, known as recovery blocks [1, 17, 27]. This is depicted in Figure 1.

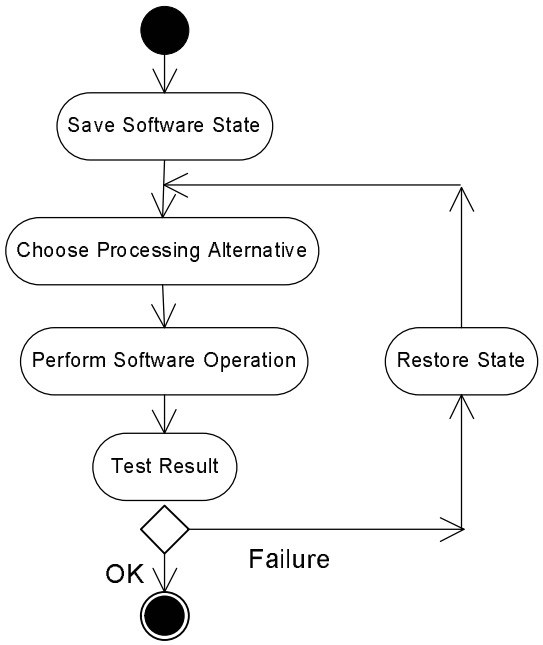

Fig. 1. A transaction (recovery block) approach to correcting for the occurrence of errors in a software system

There are two ways of returning to a previous state: (1) reloading a saved image of the system before the recently failed computation, or (2) rolling back, i.e., reversing and undoing, each step of the failed computation [4]. Both of the ways suffer from major difficulties:

1. The state of a software system might be very large, necessitating the saving of very large images
2. Many operations cannot be undone, such as those that have side-effects. Examples of these are sending a message, which cannot be un-sent, and spending resources, which cannot be un-spent. Rollback is successful in database systems, because most database operations do not have side-effects.

2.6 Compensation

Because of this, compensation is often a better alternative for software systems. As in database systems, it is often better to perform a compensating action, rather than save a checkpoint of a system with a large state. Figure 2 depicts the architecture of a robust software system that relies on compensation of failed operations.

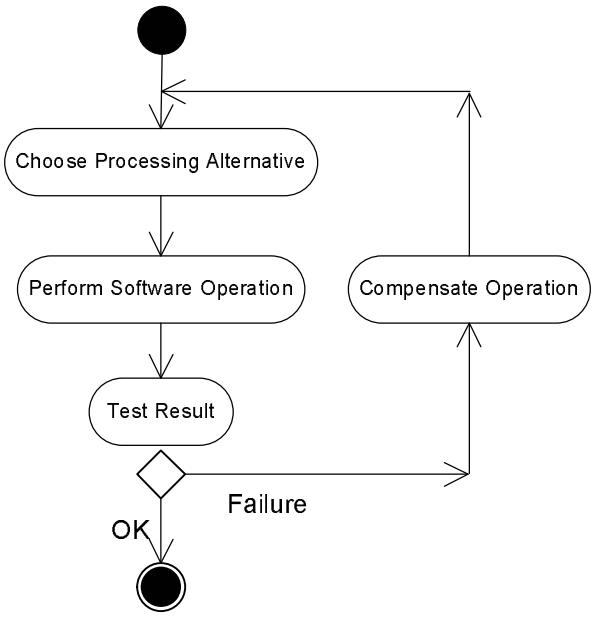

Fig. 2. An architecture for software robustness based on compensating operations

3 Architecture and Process

Suppose there are a number of sorting algorithms available. Each might have strengths, weaknesses, and possibly errors. One might work only for integers, while another might be slower but be able to sort strings as well. How can the

algorithms be combined so that the strengths of each are exploited and the weaknesses or flaws of each are covered? In solving this in a general way, we hypothesize that the end result is an "agentizing" of each algorithm.

3.1 Architectural Approaches

A centralized approach, as shown in Figure 3, would use an omniscient preprocessing algorithm to receive the data to be sorted and would choose the best algorithm to perform the sorting. Each module's characteristics would have to be encoded into the central unit. The central unit could use a simplistic algorithm for determining best, based on known facts about each of the modules. The difficulties with this approach are (1) the preprocessing algorithm might be flawed and (2) its maintenance is difficult as new algorithms are added and existing algorithms become unavailable. Also, only one module at-a-time executes, there is low CPU usage, and results are taken as-is when completed.

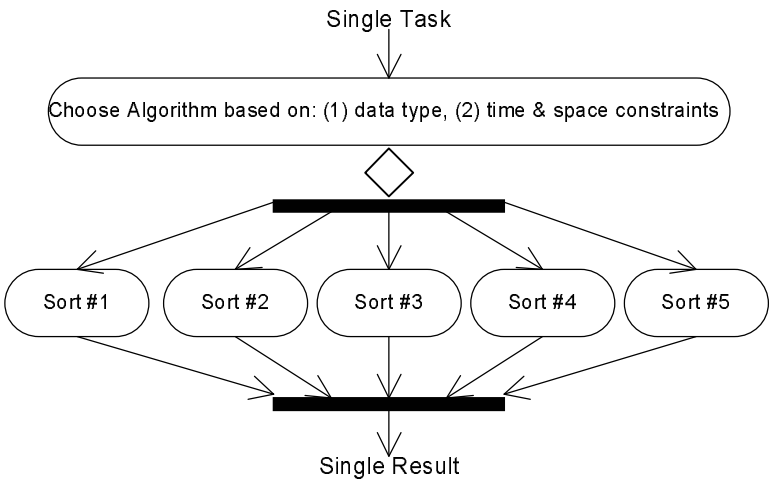

Fig. 3. Centralized architecture for combining N versions of a sorting algorithm into a single, more robust system for sorting, where a preprocessing algorithm chooses which sorting algorithm will execute

An improvement might be a postprocessing algorithm, as shown in Figure 4, that receives the results of all sorting algorithms and chooses the best result to be the output. Results have to be compared and voted on in order to determine the best. This approach is also centralized and suffers from a waste of CPU resources, because all algorithms work on the data. However, due to the comparison of outcomes, it is likely to produce better results.

A combination of the pre- and postprocessing centralized systems could also be used. Since criteria are known about each module, a subgroup could be selected to perform the desired task based on known factors such as speed, time,

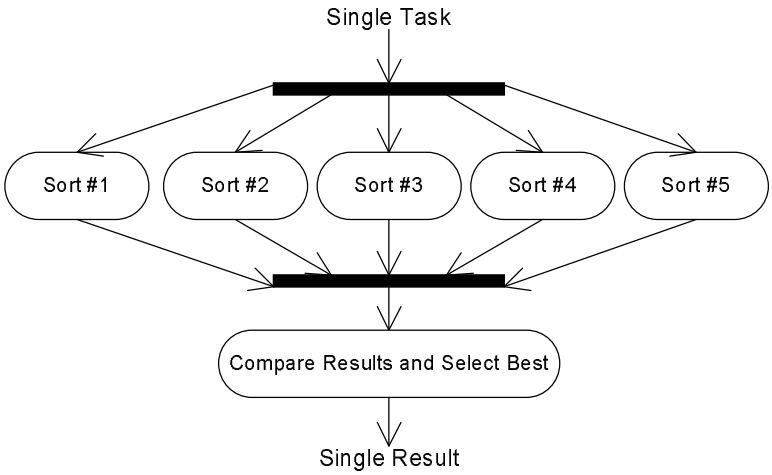

Fig. 4. Centralized architecture for combining N versions of a sorting algorithm into a single, more robust system for sorting, where a postprocessing algorithm chooses one result to be the output

and space. This subgroup would then have its results compared to determine the best results as above.

A fourth approach is a distributed solution, where the algorithms jointly decide which one(s) should perform the sorting, and if there is more than one qualified algorithm, they jointly decide on the best result. Conventional algorithms do not typically have such a distributed decision-making ability, so in this paper we investigate whether there is a generic capability that can be added to an algorithm to enable it to participate in distributed decision-making. We also show that the result has the characteristics of a software agent.

3.2 Multiagent System Approach

Moving the administrative responsibilities from the central intelligent unit and distributing those responsibilities into the different modules creates a multiagent system. An agent in this system would have to know about itself: what it needs, what it can accomplish, and how. Before accepting a task, it would have to believe it could accomplish the task. Specifically, each agent must know

- Something about its own algorithm, such as its time and space complexity, its input data structures, and its output data structures
- Something about other agents, such as their time and space complexity and reliability
- How to negotiate
- How to communicate
- How to compare results
- How to manage reputations and trust.

A transformation from the three conventional systems to an agent system would entail moving the responsibilities from a central unit, either a preprocessor, postprocessor, or combination, to the individual agents.

An agent system transformed from a conventional system based on a preprocessor approach would have the agents themselves deciding on who would do the assigned task. Since an agent knows about itself, it could choose to submit a bid, which implies an auction environment to determine task assignment. This is an acceptable approach when there are many competing agents, but robustness due to reinforced redundant involvement would be lost.

An auction environment, where agent interactions take place without any dependencies among the agents' abilities, can be represented by Lorge and Solomon's "Model A:"

$$P = 1 - (1 - p)^r$$

where the probability, P, of a group of individuals solving a problem is the probability that the group size r contained at least one individual solver, given the probability, p, of a correct solution by an individual [21]. This representation is based on the agents' being able to detect a correct solution. The result is that a group of agents will outperform an individual on a consistent basis [9].

An agent system transformed from a conventional system based on a centralized postprocessor would entail that each of the agents attempt the task and some type of voting mechanism (either with a voting factor or not) would be set up among the agents. A vote could be based on reputation only (more later) or on a majority rule vote based on a comparison of results. The communication overhead could be significant.

It should be noted that a group can sometimes be wrong, but with a functional basis, a group will be correct more often than its most accurate member. Shapley and Grofman give the following example of five weather forecasters predicting whether it will rain or not on a given day. The decision is "yes, it will rain" or "no, it will not rain." The forecasters are given weights in proportion to $log(p_i/(1 - p_i))$, where p_i is the probability of forecaster i making a correct decision. Assigning the following weights to the forecasters: 0.9, 0.9, 0.6, 0.6, and 0.6, yields a group decision correctness probability of 0.927. This is higher than that of any one individual and also of unweighted voting, which has a group probability of 0.877.

The notion of a group is determined by the number of its members, the amount of communication among the members, and the identity of the members. Infrequent communication between individuals indicates casual relationships and not a group. A group can have its own way of identifying each of its members, but the members do not cease to be individuals while in the group, because they still have a personal responsibility to themselves, their own reputation, and their own desires.

Group decision-making is uninteresting for fewer than two members, but there is also a maximum size for a group. Beyond 10 to 15 members, a group becomes an assembly (where members do more waiting around than not) or a mob (where members are out of control) [19].

The comparison of results can be done in several ways. The results can all be compared and a majority of exact outcomes would determine the results that are selected, where each of the agents have an equal chance at having their results selected. (For example, assume that there are four different results: A, B, C, and D. Five agents have result A, while two agents each have results B, C, and D. Therefore, result A is passed on as the accepted result.) This method is fine when there is a clear-cut winner, as above, but in the following case this methodology becomes cloudy. (For another example, assume three agents have result A and three agents have result B, while two agents have result C and D, respectively. A and B have the same number of votes, so an additional procedure must be used to choose between them.) A secondary factor might be information the agent knows about itself, such as whether it completed the task or not, its time and space requirements, and the total number of runtime errors it has produced in the past.

4 Initial Experiments

We collected a number of sorting algorithms, each written by a different person and therefore having different input and output signatures and performance characteristics. We converted each algorithm into a sorting 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 can sort, the data type it produces, its time complexity, and its space complexity. The sorting algorithms were written in Java and the wrappers in JADE [14].

Figure 5 is the AUML diagram of the protocol used. The sorting system begins by a notification sent to the Initiator agent about data to be sorted, which notifies the sorting agents. Upon receiving data to be sorted, each agent determines whether or not it can sort it successfully (based on the type of the data and its own knowledge of what types it can sort). If the agent believes it can sort the data, it broadcasts an INFORM message to every other agent specifying its intention, along with a measure of performance for its algorithm (based on time and space complexity).

The decision of which agent (i.e., algorithm) to choose among those that are capable of sorting the input data is made in a distributed manner: upon receiving the INFORM messages from other agents, each agent compares its own performance measure against those received in the messages. If the agent has the best performance measure, it will run its algorithm and send the results back to the system. If it does not have the best performance measure, it will do nothing. Also, once they receive the data to be sorted 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 not able to sort the data.

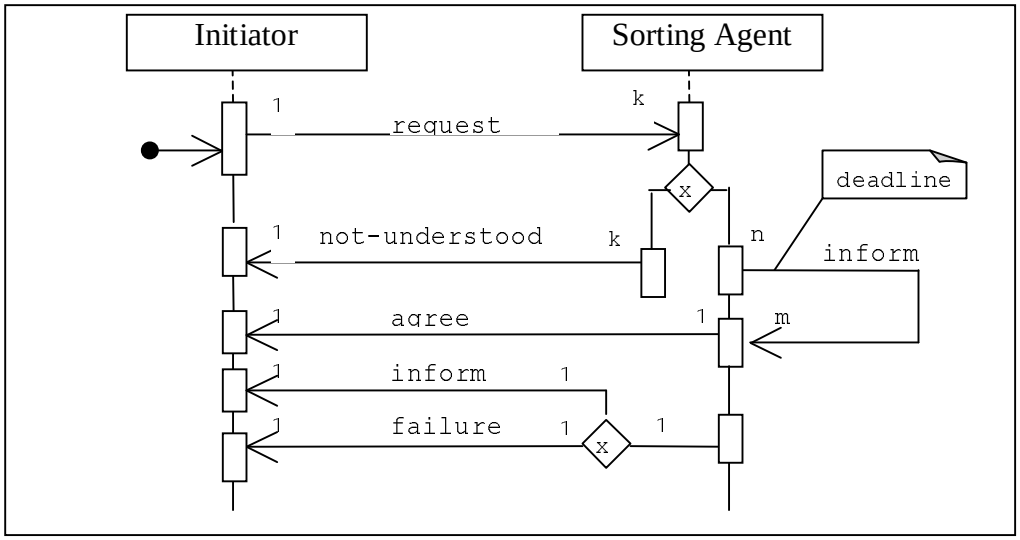

Fig. 5. AUML diagram of the agents' interactions

As can be seen, the current implementation of the wrappers is a preprocessing approach. Agents themselves decide on who will do the task placed before them. Even with the simplicity of the current implementation, results showing improvement in robustness due to redundancy were obtained. As a group, the agents sort data better than any one of them alone.

Table 1 summarizes the algorithms collected and information about them. The Additional Restrictions column shows restrictions not used by the default Wrapper to determine whether the agent can sort the data. Nevertheless, it is possible to cope with these cases by providing an implementation of a WrapperRestrictions interface, which only has one method that takes as input the data to be sorted and returns a Boolean value indicating whether the sorting algorithm can sort the data. This provides a way of customizing the wrapper for algorithms that need additional considerations, resulting in better performance for the whole system.

Finally, Table 2 provides a summary of the initial tests performed with the system. For the first set of inputs all the algorithms but RadixSort were appropriate. C.A.R Hoare's Quick Sort algortihm was selected because it had the best performance. For the second set of inputs only the QuickSort algorithm was selectable, because it was the only one that could handle strings. In the runs it was selected only when appropriate. For the third set of inputs RadixSort was selected. For the last set of inputs only the QuickSort algorithm could be selected because it was the only one that could handle reals.

As a group, the agents sorted data better than any one of them alone. Both C.A.R Hoare's Quick Sort and HeapSort algorithms could not handle inputs 2 and 3. The RadixSort algorithm could handle only one of the data sets. Finally,

Table 1. Input data type and additional restrictions for the sorting algorithms available

Algorithm	**Characteristics/Restrictions Used by Default Wrapper**	**Additional Restrictions**
C.A.R Hoare's Quick Sort	Input data type: int array (positive and negative numbers accepted)	None
HeapSort	Input data type: int array (positive and negative numbers accepted)	None
QuickSort	Input data type: Byte array Short array Integer array Long array Float array Double array String array Char array	None
RadixSort	Input data type: int array	Only 10 inputs accepted

although the QuickSort algorithm was able to handle all the inputs, it did not have the best performance.

5 Group Decision Making

The system sends data to be sorted to the group of sorting agents. For simplicity, we assume that no matter what kind of data is sent, there is at least one agent that can sort the data. Each agent receives the data, decides whether it can sort the data, and broadcasts its decision. This first step will then determine the subgroup size. Because at least one agent can sort the data, the group size will range from 1 to N, where N is the total number of sorting agents in the group.

When only one agent indicates it can sort the data, the decision is trivial. When there is more than one agent with the ability to sort a data input, then a decision strategy must be put to use to determine which output to accept.

A voting strategy is considered for our experiments. There are three basic voting incentives: the data, reputation, or secondary factors. If the strategy involves the data, then direct comparisons need to be made about the data. An agent could compare another agent's result, which was positive, with its own. An array could be used to tabulate the results for each agent based on whether there is an exact match with itself or not.

Table 2. Initial results for wrapping sorting algorithms with agents

Data Input	Algorithm Selected	Data Output	Comments
12, 45, 3, 2, 56	C.A.R Hoare's Quick Sort	2, 3, 12, 45, 56	
ann, john, sue, marie	QuickSort	ann, john, marie, sue	Only this algorithm could handle strings
9, 8, 7, 4, 3, 2, 12, 4, 5, 10	RadixSort	2, 3, 4, 4, 5, 7, 8, 9, 10, 12	Best performance but only worked for 10 inputs
3.54, 90, 23.4, 3.55, 60, 60.1	QuickSort	3.54, 3.55, 23.4, 60, 60.1, 90	Only this algorithm could handle reals

For example, assuming only sortAgents 1, 2, 4, and 6 are capable of sorting the data, the array for sortAgent1 could be as follows:

$$compare = Array(sortAgent2 => yes, sortAgent4 => no, sortAgent6 => yes)$$

Counting itself, sortAgent1 would receive three positives and one negative.

Comparisons could also be normalized according to the amount of similarity between different agent's results. Comparison could have a direct relationship between exact locations, a relationship between substrings, and/or a relationship between the number of location matches or exact substrings.

CPU usage could be reduced if the result (yes or no) of a comparison is calculated only once, instead of by both agents. Logic could also aid in a reduction of CPU usage. If an agent A agrees with an agent B exactly, and agent A also agrees with agent C exactly, then agent B and agent C also agree. The sortAgent with the highest percentage of positives is selected. A tie can have two possibilities: (1) when the agents in the tie are a perfect match, then it does not matter which is chosen, or (2) when the agents are not a perfect match, then deciding is a problem.

If the strategy involves reputation, then a single sortAgent's vote is just its reputation factor; the result would be determined based on this factor alone. A tie here is a problem. Note that a reputation can be computed by many different means: e.g., neural nets, Bayesian networks, expert systems, or a simple normality based on successes. Reputation requires feedback for the system, either internally among the agents or externally from a user.

Secondary factors, such as what the agent knows about itself, can affect a sortAgent's vote. Completion of the task has to be taken into consideration. Time and space constraints and runtime errors encountered during execution could be sorted in relative order of importance. A high number of runtime errors could affect a vote; a long delay or large space requirements could also affect a vote. A formula based on its time and space constraints and any runtime errors could be formed and used. A tie could be broken by hierarchically organizing each mitigating factor as above.

Combinations of these three basic voting incentives could be used. First, a formula based on secondary factors would be used (since the presence of runtime errors is a very important fact); second, the data comparison would be made; and finally, a normalized reputation factor based on past successes in a group would be used. For example:

```
decisionVote =  a * Completed? +
                b * number of runtime errors +
                c * time used +
                d * space used +
                e * exact comparisons +
                f * reputation (if there is one)
```

Variables a, b, c, d are secondary factors, variable e is a data factor, and variable f is a reputation variable. The "best" algorithm is determined by

$$Performance \times Flexibility \times Reliability$$

where *Performance* is a function of time and space complexity, *Flexibility* is a function of how broad a range of input and output data structures the algorithm can handle, and *Reliability* is a measure of how well the algorithm can avoid runtime errors and exceptions.

6 Conclusion: Challenges and Implications for Developers

Producing robust software has never been easy, and the approach recommended here would have major effects on the way that developers construct software systems:

- It is difficult enough to write one algorithm to solve a problem, let alone n algorithms. However, algorithms, in the form of agents, are easier to reuse than when coded conventionally and easier to add to an existing system, because agents are designed to interact with an arbitrary number of other agents.
- Agent organizational specifications need to be developed to take full advantage of redundancy.
- Agents will need to understand how to detect and correct inconsistencies in each other's behavior, without a fixed leader or centralized controller.
- There are problems when the agents either represent or use nonrenewable resources, such as CPU cycles, power, and bandwidth, because they will use it n times as fast.
- Although error-free code will always be important, 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.

– In some organizations, software development is competitive in that several people might write an algorithm to yield a given functionality, and the "best" algorithm will be selected. Under the approach suggested here, all algorithms would be selected.

Ultimately, the production of robust software will require that we understand the relationship between

– the social world as represented by humans and their physical environment, and
– the social world as represented by agents and other automated systems.

Acknowledgements

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

References

1. Anderson, T. 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.
2. Avizienis, Algirdas: "Fault-Tolerant Systems." *IEEE Transactions on Computers*, 25(12), pp. 1304–1312, 1976.
3. Avizienis, Algirdas: "Toward Systematic Design of Fault-Tolerant Systems." *IEEE Computer*, 30(4), pp. 51–58, 1997.
4. Chandy, K.N. and C. V. Ramamoorthy: "Rollback and recovery strategies for computer programs." *IEEE Transactions on Computers*, June 1972, pp. 59–65.
5. Coelho, Helder, 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.
6. Cox, Brad J.: Planning the Software Industrial Revolution. *IEEE Software,* (Nov. 1990) 25–33.
7. DeLoach, S.: "Analysis and Design using MaSe and agentTool." In *Proc. 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, 2001.
8. Eckhardt, D.E., and L.D. Lee: "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors." *IEEE Transactions on Software Engineering*, SE-11(12), pp. 1511–1517, 1985.
9. Hasling, John: *Group Discussion and Decision Making,* Thomas Y. Crowell Company, Inc. (1975).
10. Holderfield, Vance T. and Michael N. Huhns: "A Foundational Analysis of Software Robustness Using Redundant Agent Collaboration." In *Proceedings International Workshop on Agent Technology and Software Engineering*, Erfurt, Germany, October 2002.
11. Huhns, Michael N.: "Interaction-Oriented Programming." In *Agent-Oriented Software Engineering,* Paulo Ciancarini and Michael Wooldridge, editors, Springer Verlag, Lecture Notes in AI, Volume 1957, Berlin, pp. 29-44 (2001).

12. Iglesias, C. A., M. Garijo, J. C. Gonzales, and R. Velasco: "Analysis and Design of Multi-Agent Systems using MAS-CommonKADS." In *Proc. AAAI'97 Workshop on agent Theories, Architectures and Languages*, Providence, USA, 1997.
13. 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.
14. *JADE: Java Agent Development Environment,* http://sharon.cselt.it/projects/jade.
15. Jennings, Nick R.: "On Agent-Based Software Engineering." *Artificial Intelligence,* 117(2) 277–296 (2000).
16. Juan, T., A. Pearce, and L. Sterling: "Extending the Gaia Methodology for Complex Open Systems." In *Proceedings of the 2002 Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 2002.
17. Kim, K.H. 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*, 38(5), May 1989, pp. 626–636.
18. Kinny, David 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—Proc. Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, Berlin, 1997, pp. 1–20.
19. Light, Donald, Suzanne Keller, and Craig Calhoun: *Sociology* Alfred A. Knopf/New York (1989).
20. Littlewood, Bev, Peter Popov, and Lorenzo Strigini: "Modelling software design diversity—a review." *ACM Computing Surveys*, 33(2), June 2001, pp. 177-208.
21. Lorge, I. and H. Solomon: "Two models of group behavior in the solution of Eureka-type problems." *Psychometrika* (1955).
22. Nwana, Hyacinth S. and Michael Wooldridge: "Software Agent Technologies." *BT Technology Journal,* 14(4):68-78 (1996).
23. Odell, J., H. Van Dyke Parunak, and Bernhard Bauer: "Extending UML for Agents." In *Proceedings of the Agent-Oriented Information Systems Workshop*, Gerd Wagner, Yves Lesperance, and Eric Yu eds., Austin, TX, 2000.
24. Padgham, L. and M. Winikoff: "Prometheus: A Methodology for Developing Intelligent Agents." In *Proc. Third International Workshop on Agent-Oriented Software Engineering*, July, 2002, Bologna, Italy.
25. Paulson, Linda Dailey: "Computer System, Heal Thyself." *IEEE Computer*, (August 2002) 20–22.
26. Perini, A., P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos: "A knowledge level software engineering methodology for agent oriented programming." In *Proceedings of Autonomous Agents*, Montreal CA, 2001.
27. Randell, Brian and Jie Xu: "The Evolution of the Recovery Block Concept." In *Software Fault Tolerance*, M. Lyu, Ed., Trends in Software, pp.1–22, J. Wiley, 1994.
28. Wooldridge, M., N. R. Jennings, and D. Kinny: "The Gaia Methodology for Agent-Oriented Analysis and Design." *Journal of Autonomous Agents and Multi-Agent Systems*, 2000.