A Foundational Analysis of Software Robustness Using Redundant Agent Collaboration

Vance T. Holderfield[1] and Michael N. Huhns[1]

Department of Computer Science and Engineering, University of South Carolina
Columbia, SC 29208 USA
`http://www.cse.sc.edu/~huhns`
`{vance, huhns}@sc.edu`

Abstract. This paper describes an investigation into how the robustness of a software system can be improved via the use of software agents as a fundamental component of the system. Robustness can be achieved through redundancy, and we hypothesize that agents are an appropriate unit for adding redundancy. As part of our investigation, we asked 53 students to independently build one agent for a simple Multiagent domain. We did a comparison study between the different student's agents while working with each other independently and while working with each other as part of collaboration. A simple decision-making mechanism was also introduced and compared. This influx of redundancy in the collaborations provided a discernible increase in productivity. The results shown here should lead to further rich and enlightening research into agent-oriented software engineering.

1 Introduction

Agent systems, and particularly Multiagent systems, have come to the forefront of research for today's computer engineers and computer scientists [11]. Agents are cooperative, persistent, and aware [1] as well as autonomous (independent). These characteristics make agent-oriented software systems an ideal mechanism for handling the ever-growing distributed environment of today. A radical shift in our present methodologies for software engineering is due [2]. Previous software engineering methods in which software was developed for an independent closed system will not apply in the world of the Internet. Software needs to be able to handle the unexpected; in a closed or isolated system, circumstances can be mimicked and the boundaries of the expected can be thoroughly tested and researched. The vastness and complexity of the Internet makes it impossible to cover even a small fraction of the possible avenues of discourse. Researchers are looking into developing software systems that can exhibit human-like behavior, that is, capable of independent and cooperative procedures in order to perform some task or set of tasks. Agents seem to be the correct level of abstraction in which to attain this task[8].

In order for agents to be accepted as a viable alternative for programming in a distributed environment, confirmation of their reliability needs to be explored.

Acceptable software systems need to be correct and robust, so before any new methodologies can be introduced, these issues need to be addressed. If something is said to be correct, then it conforms to accepted standards, where the software designer and the user set the standards. Robustness is defined as strong and stoutly built, able to withstand the rigors of normal wear and tear. Robustness achieved through prior software engineering paradigms needs to be adapted to take full advantage of a software agent's characteristics [9].

A basis for robustness is redundancy, but the robustness of a software system will not be increased if redundancy is obtained by adding identical components. Identical components will have identical flaws and will fail at the same time and in the same ways [6]. Robustness seems to rely on heterogeneous redundancy, that is, components that have the same purposes or outcomes but achieve them via different means. If the heterogeneous components are agents, then two obvious questions that arise are "where do the heterogeneous agents come from?" and "how are decisions made if the agents have different conclusions or outcomes?"

A foundational step in the development of a new software methodology is the acceptance of group decision-making among the agents in a system. A group of agents, working together, can create decisions that are more correct and also more robust. As in human society, the correctness of a decision is not absolute; social evolution dictates the changing mores or correctness of a society [10]. Not all agents are created equally, and there is no way of assuring ourselves of the validity of an agent if we are not personally responsible for it, and in a completely distributed environment we could not be unless it was our own agent. A group of agents could overcome the possible fallacies of one or more agents by consensus with the other members of the group. Therefore, a group of agents working together for a common decision should be more correct. Being able to add members or remove members without affecting the overall decision-making process is a nod in the way of being robust. It is the group decision-making concept, albeit redundant, of agents in a distributed environment that form the basis of this research study. The aim of this paper is to prove that the individually created agents created for this experiment will have enough diversity to overcome any shortcomings by way of collaborations. This proof should provide the direction for further study.

1.1 Problem Discussion

Designing an in-house system for work on a particular project allows us the luxury of being in control of our own destiny. We as software engineers can control the system by saying how the parts of the system work together. This has been true since the early days of programming. Software engineers design systems to work as smoothly as possible. This is not saying that software errors do not occur, as they occur very frequently; it is saying however that the design of the software is in the hands of the computer scientists and software engineers who created it.

A completely procedural program, for example, has parts that are coordinated to work together. This is true whether an individual or a large team designed the software. Each procedure has specifications that have to be followed in order for the procedure to work in the scope of the program. The designers work on their 'part' of the program knowing that the 'parts' of the program providing the input or the 'parts' their procedure is providing output to are being completed a certain way, even if they do not know the exact details of the other's methods. There is a trust that is developed between the different designers/programmers of the procedures. The simplest case is of an individual who is working on a program alone. He trusts himself to complete a procedure that would work hand-in-hand with the other procedures that he had developed. Teams of programmers trust that competent programmers are developing the other parts of the overall program. Again, this is not to say that errors or bugs do not occur, in fact, it is expected. The designers/programmers then work to correct the errors as best they can.

Even in object-oriented programming where programmers reuse on hand classes (for reuse is an attractive characteristic in object-oriented programming), being able to control the system being developed is an important consideration when developing a software system. Protocols have been developed and maintained to ease the mind of system designers when they use the work of other individuals.

Software systems have been developed this way since the beginning. A part or parts of a system relying on other parts to work properly is an important criterion in developing these systems. Then to have the software be robust, that is to be strong and ably built, the software needs to be able to accomplish its task without any side effects affecting the outcome. The burden lies on the designer/programmer to develop a robust system, where all the different parts of the system work together. People, however, are not perfect, and they do not create perfect software systems or even perfect designs for software systems. Who's to say that one computer scientist's design is any better than any other? The relevance of whose design is better is solely based on the situation requiring the design; it is probably more relevant to say 'whose software design is more appropriate in this situation?' But as computer scientists, we have to deal with our own knowledge and/or that of our coworkers in assessing the situational existence of a certain software design scheme.

There are often many ways to accomplish a task. In an ideal situation, a computer scientist would incorporate many different methods for solving a task into a single system, being able to take advantage of differences among various methods to solve a problem based on the different situations that may arise. Utilizing these differences to solve potential situational problems would make the system more dynamic and therefore more robust. The system would be able to react to different situations in different ways based on the different methodologies bred into the different designs. Is this type of method integration possible? It is possible, but at a great expense. Hardware considerations have become less of a concern at present, although they are still a factor. The primary concern would be on garnering the different design algorithms for the system. Being a single designer, how possible would it be to come up with enough different design algorithms to take advantage of the ability for a more dynamic behavior? It would be a difficult undertaking for a single computer scientist to design different algorithms for the same task. Even a team of designers would encounter many difficult dilemmas in trying to come up with different algorithms and then coordinating the resulting software components to work together.

As mentioned in the introduction, software agents seem to be the correct level of abstraction for handling such a task and with the growing popularity of the Internet, the problem of finding different designers for coming up with different algorithms also seems to be solved. Although the application of using agents to redundantly work together does lend itself for a closed system, the openness of the Internet allows that an open system would take advantage of the above-mentioned ideals. The idea is for independent design algorithms to be used. Designers would not have to be in the same locale or even to know about each other in order create software agents. As long as each software agent can complete its task appropriately, the 'how' is unimportant.

1.2 Background History

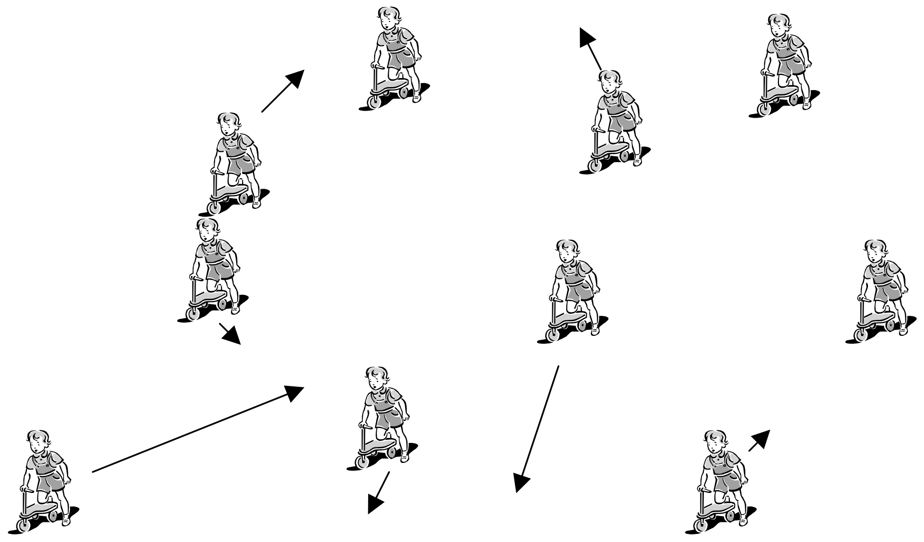

Fig. 1. Children (and autonomous agents) can be a robust circle-forming algorithm.

This experiment is an extension of an idea from Huhns [7] and a first implementation of it by Glenn Lawyer. The idea of a circle of agents is based on young children who are just old enough to be able to comprehend and to follow the instruction of forming a circle while playing on a playground, as depicted in Figure 1. When asked, children can form a circle with a decent shape and relative distance between each child. This idea forms the basis that with limited instruction and control, young children, who understand the general idea of a circle, can on their own perform what was asked of them. The children are without any type of hierarchical leadership and each child is a completely separate entity capable of determining where and how to form this circle with the others. The children are adaptable; the circle can be increased to allow for the addition of children or decreased when a child leaves the circle. The children can also vary the location of their circle based on where most of the children are located. These concepts can be applied to software systems, particularly an agent-oriented system and was done so in the demonstration program.

The original program, when executed, displayed a rectangle depicting the field for the circle agents. Agents could be added or removed via 'Add' and 'Remove' buttons. A 'Line' button caused the agents to form a line, a 'Circle' button caused the agents to form a circle, and a 'Center/Roam' button caused the agents to move toward the center or roam randomly throughout the field.

The agent size, which could be one of four values, had a bearing on the circle and/or the line being formed. The agents when forming a circle and/or a line would start by finding the center of mass of all the agents located inside the agent field. That is, the average location or center of all the agents. The bigger the agent the more 'weight' it carried in determining the center of mass. After the agents were placed inside the field, they would begin to roam randomly. By pressing the 'Circle' or 'Line' button, the agents would begin to move into the appropriate formation.

To describe how a number of developers could independently contribute agents to this experiment, we need to be precise about the characteristics of the software. The original circle agent demonstration program was made up of two separate Java files: IntelliField.java and IntelliAgent.java. IntelliField consisted of the main() subroutine that created the GUI display. It was responsible for capturing the mouse events (buttons pressed, size selected, location to add or remove an agent) and instantiating agents, by way of IntelliAgent. The IntelliField was also responsible for drawing each agent. The IntelliField performed all the calculations involved in forming a line and forming a circle, including, but not limited to, finding the furthest agents left or right in order to form a line, finding the center of mass to determine the center of the circle and also calculating the radius of the circle. The IntelliField also determined the spacing between the agents (which was equal).

An instantiation of IntelliAgent represented each dot (agent) on the screen. IntelliField performed the instantiation with information gathered from the user's preferences (buttons pressed, size selected, and location). The IntelliAgent was responsible for setting the appropriate size and color (as requested) and also for moving the agent. The IntelliAgent used methods located in IntelliField to find out where its own instantiation should go. The agent would then move in tiny increments toward the goal set up by the IntelliField.

1.3 Assignment

The original implementation was only a demonstration program. The IntelliField class of the original demonstration program performed all the calculations and told the agent where it was to go, be it a line, a circle, or just to roam randomly. This deprecated the agent from being autonomous. The original files needed to be adjusted in order to make the demonstration program more agent-like and remove unnecessary variables. All agents were made the same size and their only goal was to form a circle. Agents were added at random locations, rather than at the location of a mouse click. The result was a better basis for research.

The calculations for arranging the individual agents into a circle were moved from the IntelliField class into the agent. Drawing the agent was moved to the individual agent as well. The IntelliField became an environment that keeps up with the location of each agent and the total number of agents.

An assignment was given to 53 graduate students in computer science and engineering to construct an agent for the above-mentioned circle demonstration program. The typical student's level of expertise was of various class-type projects in C++ or Java. This was the first agent-oriented programming experience for most, if not all, of the students. The demonstration program was created with Java to enable the students to use threads and a GUI. Each student created his or her own IntelliAgent class, which was designed so that its agent instantiations would only be guaranteed to work with other agents of its own kind. The students were provided with a fully functioning IntelliField class, and were able to use methods located in the IntelliField class. The methods allowed a student's instantiated IntelliAgent access to the total number of agents and the location of all instantiated agents. A method was also provided to allow the instantiated IntelliAgent to update its own location to the IntelliField. The students were encouraged to be innovative in how to make their version of the circle demonstration program work.

Each student's IntelliAgent was tested and analyzed. Three students manipulated the IntelliField to the point where their IntelliAgent was not usable. In order to be able to use the different IntelliAgents, the same IntelliField needs to be used; otherwise, different IntelliFields would have to be created for each of the IntelliAgents. This stipulation was required when the assignment was given. Of the fifty remaining IntelliAgents, five different algorithms were used in order to perform the required function: forming a circle. Most of the students used the method from the IntelliField to acquire the location of all the other IntelliAgents and the total number of IntelliAgents to calculate the center of mass (averaging the x and y coordinates). Using this center of mass, the virtual center of their circle was found. A few students used specific locations on the agent field as the center of their circle, such as the upper left corner or the absolute center of the agent field. The radiuses were calculated in different ways also. The most common method of calculating the radius of each individual student's circle of agents was to assign a minimum length (or radius) that grew dynamically with the addition of more agents, therefore the more agents that were added, the larger the radius and consequently the larger the circle. The other method that was used to calculate the radius of the different student's circles was to hard code a given radius. The radius was the same no matter how many agents were involved. A typical IntelliField/IntelliAgent specification is shown in Figure 2 and the code for a sample IntelliAgent is located in the Appendix.

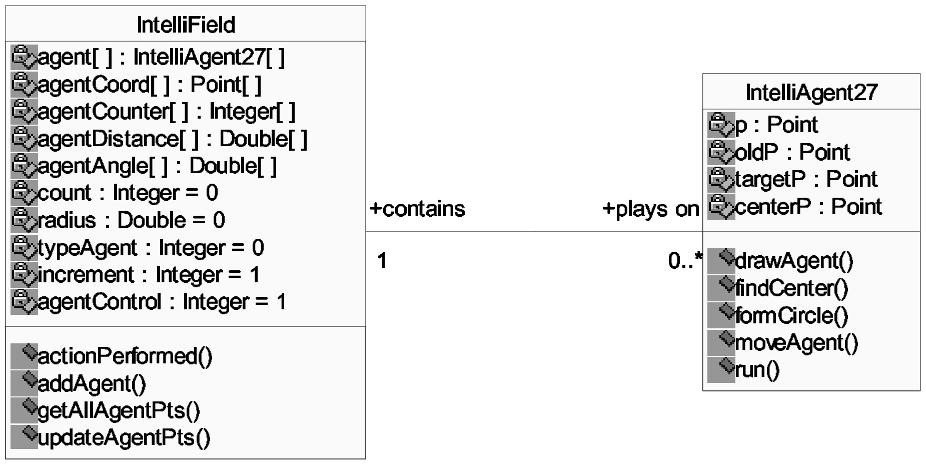

Fig. 2. Class diagrams show the abstract specifications for a typical IntelliField and IntelliAgent.

Several students introduced other innovations. Some students equalized the spacing of the agents along the radius of the circle, which was not a criterion of the assignment. The most basic way was to pre-select a location for the newly added agent to aim for in getting in place on the circle. An added agent would strive for a certain angle, in this case it was the positive x-axis of the x-y coordinate system using the center of the virtual circle as the origin of the coordinate plane. The agent no matter where it was introduced would proceed to this location, the other agents would then calculate their distance between each other based on the newly added agent (this was actually a calculation of the angles, as the number of agents in the field would be divided into the total degrees of a circle and geometry would be used to calculate the new location for each agent) and begin to move to their new location on the rim of the circle while the newly added agent would proceed to its designated spot. A more adventurous undertaking that a couple of students decided to use was to make the newly added agent go the nearest location on the circle. A new point of reference had to be calculated to determine the location of each of the agents.

2 Methodology

Each student created his or her version of the IntelliAgent to work only with instantiations of itself and the preexisting IntelliField. For this experiment, the different IntelliAgents would be assimilated into a single system to work in combination with each other. Groups of IntelliAgents would be formed to collectively drive the actions of a single child representative. These groups on average should be able to collectively work with other groups of similar construction to better form the geometrical shape of a circle.

After gathering the students' different IntelliAgents a method had to be devised to perform analysis on the system, including the IntelliField and the different IntelliAgents. The IntelliField was adjusted to accommodate the different IntelliAgent instantiations, and each IntelliAgent was given a unique identifier from 00 to 49. A method was introduced to the IntelliField to allow for the collaboration or group decision-making among different agents in controlling a single child representative. The method for collaboration was to average the x, y-coordinates of each IntelliAgent in a group and set this average as the new location for the child representative. It was these different child representatives, each controlled by a different collaboration that would form the required circle.

Since each student calculated the center of mass to help to establish the center of his or her circle, the standard deviations of the different collaborations from this center were calculated for all of the collaborations. Methods were introduced to the IntelliField to calculate the new center of mass, the radius (based on average distances of the collaborations), and to calculate the standard deviations.

An interactive session at the beginning of execution of the system allowed the user to determine the criteria for each particular trial. The user was asked which IntelliAgent to begin with (from the numbers 00 through 49), what collaboration structure to use (how many agents in a single collaboration), and also an increment criterion (this allowed for a more random selection of collaboration partners). A typical dialogue would be as follows:

System: *Which agent would you like to start with?*
User: *5*
System: *What control structure do you want to use?*
User: *3*
System: *What increment would you like to select agents?*
User: *2*

In this situation, the first agent instantiated would be IntelliAgent05, there would be three agents in a single collaboration, and the IntelliAgents inside each collaboration would be two apart or in other words: IntelliAgent05, IntelliAgent07, and IntelliAgent09. This setup of the IntelliField with each of the different IntelliAgents makes it easy to analyze many different situations. The control groups for this experiment are as follows:

1. Single agent control in a homogeneous system
2. Single agent control in a heterogeneous system
3. Collaborative agent control in a heterogeneous system

A final subgroup was setup from the third control group to demonstrate the possible advantages of decision-making among the agents. As was stated earlier, the agents in this simple environment merely took an average of where each agent decided to move. Since the center of mass was to be the origin of the circle, the simple method of ignoring the farthest outlier made this system a best-four-out-of-five system. On each move, the 'best' four agents (based on distance from the origin) were used to place the child representative. All five agents were then updated accordingly based on this data.

3 Results

In this experiment, the different control groups and subgroups mentioned above were each tested with many trials and the average of their standard deviations and circle radii was calculated. The data was then normalized based on the standard deviation and radii. The normalized distances based on radii provide a better comparison among the different systems.

In the first control group, single agent control in a homogeneous system, fifty trials were run according to the fifty different students. A single trial consisted of a single instantiation of an IntelliAgent that worked to form a circle with other identical instantiations. The average standard deviations for the different trials were either zero or extremely negligible. The second control group, single agent control in a heterogeneous system, consisted of trials where a different IntelliAgent controlled a single child representative. The third control group, collaborative agent control in a heterogeneous system, was further divided into subgroups based on the number of IntelliAgents in the collaborations. The subgroups were with two member collaborations, three member collaborations, four member collaborations, and five member collaborations. Twenty-five trials were run on the second control group and each of the four subgroups in the third control group. The data shows that the third control group with collaborative control gained an improvement on average from the single agent control system of the second control group. The last subgroup, best four-out-of-five best fit in a heterogeneous system, showed the most improvement on average compared to the previous heterogeneous systems (see Table 1).

4 Discussion

Since most of the students' IntelliAgents depended on the center of mass principle discussed earlier, the agents forming the circle were generally all affected by one another. A single 'stubborn' or 'stupid' IntelliAgent who did not follow the norm of its fellow IntelliAgents would tend to influence the location of the circle in its favor. Multiple 'stubborn' or 'stupid' IntelliAgents could further compromise the relative size of the circle as well as its location. This experiment demonstrates that the circles formed were based on input from all the IntelliAgents in a group, whether it be one or five.

Table 1. The average standard deviations using ten child representatives of collaborating agents shows a marked increase over non-collaborating agents. Fifty trials were used for the single agent, homogeneous system. Twenty-five trials were used for the rest of the systems.

Agent System	**Total Different Agents**	**Average σ**	**Average Radius**	**Normalized Distances in Radii**
Single Agent Control, Homogeneous System	10	$\sim$0	48.74	0
Single Agent Control, Heterogeneous System	10	93.44	89.11	1.05
Two Collaborating Agents, Heterogeneous System	20	41.20	55.23	0.83
Three Collaborating Agents, Heterogeneous System	30	39.16	53.46	0.73
Four Collaborating Agents, Heterogeneous System	40	39.44	54.21	0.73
Five Collaborating Agents, Heterogeneous System	50	40.68	56.83	0.72
Four-Out-of-Five Best-fit, Heterogeneous System	50	31.12	49.93	0.62

The results of this experiment show a definite positive reaction to multiagent collaboration in the third set of control groups as compared to the single agent control of the second control group, and even more so when a simple decision-making mechanism was put in place. While the results are not as altruistic as the negligible standard deviation found from group one, they are encouraging enough to elicit further study. The comparisons show that in the closed simple society created for the purpose of this experiment, improvement can be had over a completely heterogeneous system of agents. The best-case situation is for the IntelliAgents to agree (i.e., an average) to be exactly where the child representative should be: the radius distance from the center of the circle. A worst-case scenario is for all members of a group to disagree badly, and the agent would move haphazardly. But on average, as shown, the standard deviations of the collaborative groups were better than that of the heterogeneous system with a single IntelliAgent control group.

This experiment also showed the diversity of the students and their agents. Simple agents in a simple environment, even with a simple decision-making mechanism, outperformed a system of agents acting individually. Although some agents contained errors and all performed their tasks differently, collaboration of different agents compensated the group as a whole. The best four-out-of-five decision-making mechanism only further demonstrated the enormous potential of redundant agent group collaborations, where badly behaving agents could be excluded entirely.

The limits of this experiment are in its simplicity, such as the individual components comprising the experiment were created by students fulfilling a class assignment, some with little or no knowledge of agents or Java programming. The system had to be kept simple and easy enough for the students to comprehend and complete it in a reasonable amount of time. Therefore, the system was designed to accomplish a fixed task in a static environment. Some of the IntelliAgents created by the students had minor flaws that could be caught on an individual level, yet could cause the system to provide nonsensical results or to crash altogether. The results that had to be discarded for this reason were only done so because of the limits of the system that it was created for. Group decision-making on a more grand scale should be able to circumvent the possible errors of some of its group members by a more sophisticated group decision-making process, as mentioned below.

The proof of the theoretical importance of this experiment is abundantly clear. Agents working together to reach consensus can often outperform individual agents working independently. In a static environment such as the one in this experiment, the value of this statement is not as clear, but in a completely distributed environment such as the computing world is becoming, the value is immense.

5 Future Considerations

This project is a foundational step into many different avenues for further research, as described next. Whether the system is closed or open should make no difference, although an open system would appear to add to the difficulty of the research.

5.1 How Groups Make Decisions

The idea of group-decision making itself has many different possibilities. How does a group of agents work together to come up with a group-oriented decision? Note that a group of agents working together in the sense of this research is different from the notion of agent teams, who work together to fulfill some task when it is advantageous for them to do so. In our research, a group of agents work together in order to form a more robust system. There are many different factors that help to determine how groups make decisions:

1. the group composition
2. the nature of the group's tasks
3. the degree of cohesiveness of the group
4. the number of members in the group
5. group member communication
6. decision heuristics
7. leadership (if any)[13]

Group decision-making can have many benefits. 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 were given weights in proportion to $log(p_i/(1-p_i))$, where p_i is the probability of forecaster i making a correct decision. In their example, the forecasters were given the following weights: 0.9, 0.9, 0.6, 0.6, and 0.6. Assigning weights and determining a group decision by weighted voting would generate a group probability of 0.927. This is higher than that of any one individual and also of unweighted voting, which would have a group probability of 0.877 [12].

Jurists in a trial are allowed to deliberate and discuss the evidence, and arguing and compromising can take place [4]. Applying these distinctions to agents can give a more dynamic group structure. Agents would be free to express their opinion based on their individual design algorithms and present them to their agent peers. Obviously, a more thorough communication spectrum than the simple agent communication protocols used here is needed. More extensive alternatives have been proposed to create a more suitable culture for agent communications [3]. The agents must have the ability to concede or compromise in order for the group decision to be a consensus.

5.2 Group Formation

Another area for future research is in how the groups come to be formed. In a closed system, the groups are often specified at design time. In an open system or a distributed environment, this task becomes more difficult. There are two main areas that group formation can be divided into: (1) the process by which two or more agents decide to get together, and (2) the reasons that agents would want to get together.

Intensive agent communication might be required for group formation. The same argument as in group decision-making by way of consensus could and should apply here. The agents would need the ability to argue, discuss, and compromise into possibly sacrificing some of their own individualism for the sake of the group. The agents would need to go through a period of orientation to decide how and what kind of group is formed. This period of orientation would establish the task at hand and the various rules or mores that the group will follow [5].

Why a group of agents should form is the theme of this paper. A more robust system, based not on an individual agent's algorithm, but on several agents' algorithms, would give a more dynamic system and therefore a more robust system. As stated above, a closed system could possibly take into consideration during system design as to what and which agents are grouped with other agents. An open system needs to take a sociological approach to group formation. Generally, forming a group can fall into two different categories: (1) grouping with like agents and (2) grouping with unlike agents. Forming a group with similar agents could ease the communication issues mentioned above. Like agents are more apt to have the same skills and vocabulary, which would allow for an easier transformation during the orientation phase of group formation and also during the actual decision-making stages. Possible detriments would include the possible lack of differing algorithms. If the agents are alike and their methods are alike, then their decisions are apt to be alike. If an agent combines with other agents not like it, the number of differing algorithms could actually equal the number of different agents. This would allow for a more dynamic system, but the orientation phase of group formation and the deliberating stage of the agent decision-making would increase greatly in complexity.

5.3 Agent Awareness

Agents, as described in the introduction, are aware of their environment. In order for agents to form groups, whether with like or unlike agents, and make decisions by any of the methods mentioned above, a new level of awareness needs to be added to the agent credentials. Agents would need to be able to discern other agents in order to locate suitable agents to form groups with. A distinction between 'what an agent does?' and 'what an agent is doing?' needs to be established. 'What an agent does?' would signify an agent's goal or purpose. 'What an agent is doing?' could possibly signify an agent's subgoals. To form a group, either possibility would work; an agent could form a group with an agent with the same goals or subgoals as the agent. It would definitely be easier to find an agent with same subgoals, since a particular agent's goal may be too detailed to match exactly with any other agent's goal. Looking for an agent's goal would require a degree of awareness not currently available, but to look even deeper at an agent's subgoals would prove to be an extremely difficult chore.

6 Acknowledgements

A special thanks is warranted to the many reviewers for their valuable input and comments.

References

1. Bigus, Joseph P. and Jennifer Bigus: *Constructing Intelligent Agents Using Java,* Wiley Computer Publishing (2001).
2. Cox, Brad J.: Planning the Software Industrial Revolution. *IEEE Software,* (Nov. 1990) 25–33.
3. Dignum, Frank, Barbara Dunin-Keplicz, and Rineke Verbrugge: “Dialogue in team formation: a formal approach” In van der Hoek, W., Meyer, J. J., and Wittenveen, C., Editors, *ESSLLI99 Workshop: Foundations and applications of collective agent based systems,* (1999).
4. Finin, Tim, Yannis Labrou and James Mayfield: *Software Agents,* MIT Press, Cambridge (1997).
5. Forsythe, Donelson R.: *Group Dynamics,* Brooks/Cole Publishing Company, California (1990).
6. Holderfield, Vance T. and Michael N. Huhns: “Robust Software” *IEEE Internet Computing,* Vol. 6, Num. 2 (2002).

7. Huhns, Michael N. "Interaction-Oriented Programming" *Agent-Oriented Software Engineering,* Paulo Ciancarini and Michael Wooldridge, editors, Springer Verlag, Lecture Notes in AI, Volume 1957, Berlin, pp. 29-44 (2001).
8. Jennings, Nick R.: "On Agent-Based Software Engineering" *Artificial Intelligence,* 117 (2) 277-296 (2000).
9. Lewis, Ted: "The Next 10,000 Years: Part II" *IEEE Computer,* May 1996 78-86.
10. Light, Donald, Suzanne Keller and Craig Calhoun: *Sociology,* 5th ed. Alfred A. Knopf, New York (1989).
11. Nwana, Hyacinth S. and Michael Wooldridge: Software Agent Technologies. *BT Technology Journal,* 14(4):68-78 (1996).
12. Shapley, L. S. and B. Grofman: "Optimizing group judgmental accuracy in the presence of interdependence" *Public Choice,* 43: 329-343 (1984).
13. Swap, Walter C., et al: *Group Decision Making,* SAGE Publications, Inc., Beverly Hills, London, New York (1984).

Appendix: Example Student's Code for an Agent

```
//Agent class definition
public class IntelliAgent27 extends Thread
{  Point p;                     //x,y coordinates
   Point oldP;                  //old coordinates
   Point targetP;               //target coordinates
   Point centerP;               //center of mass (agents)
  // Constructor that creates this individual agent.
   public IntelliAgent27(int agentNumber, Point coord,
                       IntelliField Field)
   { super ();
     p = coord;                 //sets point
     oldP = coord;              //sets old point
     targetP =coord;            //sets target point
   }
  // Finds the center x and y coordinate of the agents.
   public void findCenter()
   {  Point[] tempPoint = Field.getAllAgentsPts();
      int pSize = Field.getCount();
      //averages the agents location
      for (int i = 0; i < pSize; i++)
      {  centerP.setX()+=tempPoint[i].getX();
         centerP.setY()+=tempPoint[i].getY(); }
      centerP.setX()=centerP.getX()/pSize;
      centerP.setY()=centerP.getY()/pSize;
   }
  // Calculates the target coordinates for the agent
   public void formCircle()
   {  int pSize = Field.getCount();
      double DistC = 0;          //distance from center
      double ExpDistC = 5*pSize; //spacing around circle
      this.findCenter();

      //find target position distance from center
      DistC = Math.sqrt(((targetP.getX()-centerP.getX())
                        *(targetP.getX()-centerP.getX()))
                       +((targetP.getY()-centerP.getY())
                        *(targetP.getY()-centerP.getY())));
      (ExpDistC > 75)?(ExpDistC = 50 + pSize);
      if (DistC == 0)            //Target is center
      {  targetP.setX()=((ExpDistC / 2));
         targetP.setY()=((ExpDistC / 2)); }
   }
  // Draws in its new location, erases the old location
   public void drawAgent ()
   {  Graphics g = Field.getGraphics();
      int ASize = 6;             //agent size in pixels
      //Color in old dot at old position
      g.setColor(Color.white);
      g.fillOval(oldP.getX(), (oldP.getY()-ASize/2),ASize,ASize);

      // Color in new dot at new position
      g.setColor(Color.black);
      g.fillOval(p.getX(),(p.getY()-ASize/2), ASize, ASize);

      oldP.setX() = p.getX();    //keeps up with old value
      oldP.setY() = p.getY();    //to draw over with
      g.dispose();
   }
  // Moves the agents (this is done continuously)
   public void run()
   {  System.out.println("run, little agent, run!");
      moveAgent();               //moves the agent
   }
  // Calculates the new x and y coordinate.
   public void moveAgent()
   {  p.setX() += (targetp.getX() - p.getX()) / 4;
      p.setY() += (targetP.getY() - p.getY()) / 4;

      //updates the coordinates back in the environment
      Field.updateAgentsPt(agentNumber,p);
      this.formCircle();         //calculates target
}
```