Abstract

With the goal of more efficient software development and implementation, this paper shows two ways to increase the effectiveness of a task driven by an algorithm. First, during design implementation, algorithms designed to complete the task are compared, and the best algorithm is chosen. Second, during runtime, multiple algorithms, predefined for the task, are compared using a specific input, and again, the best algorithm is chosen. Both of these methods are accomplished using intelligent agents that examine each algorithm, predetermine expected results, and then broadcast these results to each other so that each agent can intelligently determine whether or not its algorithm is the best. Through this combination of multiagent systems and N-version programming, we see an increase in accuracy and efficiency as well as robustness. These agents assist in forming a basis for better algorithm development and fewer bugs during software testing. These ideas and the proof-of-concept agent this paper covers will spur on future research that should develop improved agents that will be able to provide a consensus algorithm that improves on all input algorithms.

I. Introduction

In every engineering discipline, maximum efficiency is the goal to strive for in any process while maintaining accuracy. In the field of software engineering, we look for efficiency in two areas, the efficient use of man-hours when developing a piece of software, and the runtime efficiency of the software in both time and resource management. To maintain accuracy, we must have software that is robust and fault tolerant, that is, reactive to errors that arise, especially in the complex systems that run many key elements of our society [Huhns 2003]. The problem remains that as efficiency increases, the likelihood of faults also increases, requiring extra resources to be devoted toward correcting or bypassing these faults, and thus decreasing efficiency.

One particular method of increasing the robustness of software is through the N-version programming (NVP) method [Huhns 2003]. This is a form of multiversion programming that runs through various modules for a certain task and has a decision algorithm that determines what the correct output should be. The largest problem with this lies in the amount of space and time this takes up, that is, N times what a single version of the software would be. As it has been shown, the degree to which NVP affects the efficiency of a system can be greatly reduced using multiagent systems [Turlapati 2005]. Since agents are not only able to learn from their previous results but are also able to communicate with each other, they gain significantly more information immediately. This means that while the robustness of the system is increased, the efficiency loss is minimized, in all ways, benefiting the user of the program in which this is implemented. This
paper shows the use of an agent examining an algorithm that communicates with different iterations of itself to solve the problem of determining which algorithm is the most suitable and efficient to use.

To that end, we realize that these same agents that allow a program to become more robust can be adapted in such a way as to aid the implementation side of a piece of software's life cycle. The same way that the agents choose the best algorithm among many to implement a specific input allows them to take many inputs during development, and in time, deliver the best algorithm to handle each type of input. This means that developers may make use of a major benefit of NVP while still retaining efficiency; the use of the “mail-order” concept of writing a software module [Bharathi 2003]. This will also bring about the most efficiency both during development, since human testing of all possible algorithms takes a significant portion of time, and during runtime, since the chosen algorithm(s) can now form a single version program or at least a multiversion program that is controlled by a preprocessor with the knowledge of the agents.

It should be noted that multiagent systems might not be required to be used for increasing robustness. Software is not akin to hardware, which can attain robustness in many ways, such as through component repetition, self-healing, or system models of the way the system should act [Huhns 2004]. Thus, other methods of achieving robustness in software must be examined. Taking a lesson from hardware, redundancy, whether through repetition or system models, is the most common avenue for increasing robustness [Huhns 2002]. This can be seen in software through N-version programming as has been mentioned as well as other methods such as recovery block programming [Bharathi 2003]. However, all of these have the inherent problem of decreasing efficiency, thus, any combination with multiagent systems can only benefit the process.

Many developers, however, loathe to accept the idea of leaving faults within a program, favoring instead the idea of fault prevention. Research, then, is driven to improve software engineering techniques such that faults no longer crop up, unfortunately, this is still prone to human error and is unlikely to every be completely accomplished. Thus, an automated, i.e. agent based, approach is positioned to fulfill the task of software robustness.

With software development thus moving toward redundancy and automation, it is fitting that our intelligent agent works in both these areas to solve the problem of software efficiency.

2. Approach

In order to address the problem of comparing algorithms to each other using agents, the development framework JADE (Java Agent DEvelopment) was used. JADE allows for a powerful implementation of agents in the Java programming language [JADE website]. These agents have considerable freedom for what they can do on their own through the use of specific, programmed behaviors as well as an extensive collection of message
passing and receiving abilities that are in every way compliant with FIPA regulations [FIPA website]. In the following sections, first introduced is the agent's initialization, followed by the agent's decision making, and finally, the assumptions we make for the implementation and running of the agent.

2.1 Agent Initialization

The agent itself is required to test three areas before passing its information along to each instance of itself across the platform. These areas are input type and space and time complexities. When an agent is initialized, it receives as arguments the algorithm that it is to analyze as well as the input type of the data that the algorithm would be executing. As the algorithm is starting up, it performs the first of its three tests in checking to make sure that the algorithm’s input type matches that of the data. If there is a mismatch here, the agent terminates, otherwise, the agent registers itself, waits, and then checks for all other agents that have registered as having algorithms.

2.2 Agent Resolution

The only behavior that the agent has is a stepping behavior to facilitate sending messages to each of the other agents. The first step of the behavior starts with the analysis of the space and time complexities. Once complete, the agent informs all other agents of these values. The next step receives all messages sent out and checks each algorithm’s complexities against its own. If both the space and time complexities from a received message are better than the agent’s own algorithm, then the agent terminates, sending a message to all other agents saying so. Otherwise, once the complexities from each message have been collected, the agent sends a message confirming it is still alive. The third step again involves gathering all messages. If a message is received saying that an agent is terminating, the agent removes the expired agent’s complexities from its collection. Once all messages are received, a voting algorithm then decides whether or not that agent should run based on a comparison of the complexities.

2.3 Assumptions

With this agent, quite a few assumptions are made for the sake of testing. The algorithm that each agent receives should be in a file of its own, uncompiled, and only a method instead of a class or object. Furthermore, the agent assumes that there is only one input type.

Outside of testing purposes, the agents are designed to work together on a single platform. Though JADE has the ability to work across multiple platforms, for the sake of speed and the eventual intention of use, the agent was not programmed utilizing this capability.
3. Experiments

4. Results

5. Conclusions

6. Future Work

While this agent shows that there is significant benefit to using multiagent systems in conjunction with the N-version programming theory in order to increase fault tolerance and robustness, there are still many ways to extend this experiment. The most obvious addition is that of an algorithm that satisfactorily analyzes the complexities of space and time. This is needed to make this agent usable in any fashion other than research. Along the same line, the input type checking can be improved. Instead of a simple check to make sure that the single input of a method is same as the request input type, this can be expanded to include both multiple inputs as well as checking a class or an object. Additionally, the agent should be able to recognize when the required input type is a subclass of the algorithm’s input type.

7. Acknowledgments

NSF
Huhns
REU faculty

References