A Background Layer of Health Monitoring and Error Handling for ObjectAgent

Joseph B. Mueller, Derek M. Surka, and Joy J. Lin

Princeton Satellite Systems
150 S. Washington St., Suite 201
Falls Church, Virginia 22046
{jmueller, dmsurka, jjlin}@psatellite.com

Abstract

ObjectAgent is an agent-based, message-passing software architecture that utilizes natural language processing to provide autonomous control to complex systems. As a form of distributed programming, the architecture relies on agents sharing information in order to accomplish various tasks. Because this architecture is both flexible and reconfigurable, it is a natural platform for implementing artificial intelligence techniques. Crucial to the success of such a system is its ability to detect and recover from faults, and to monitor its internal health. A layer of software has been added to ObjectAgent which enables the user to create families of agents, quickly define their health monitoring characteristics, and easily implement error detection and recovery algorithms. The result is a layer of error handling and health monitoring that runs seamlessly in the background.

Introduction

The ObjectAgent system is an agent-based real-time software architecture designed specifically for distributed, autonomous control. During the first phase of development, ObjectAgent was prototyped in Matlab and it is now being ported to C++ for demonstration on a real-time, distributed testbed and deployment on TechSat 21 in 2003.

Previous papers have addressed the basic Matlab architecture of ObjectAgent and have described the research into agent organizations for distributed satellite control [Schetter 2000a] [Schetter 2000b]. Papers have also described the basic C++ architecture [Surka 2001] as well as the application of ObjectAgent to the TechSat 21 program [Zetocha 2000]. This paper focuses on the recently added functionality of health monitoring and error handling, and how these features complement organized networks of agents. The paper is organized as follows:

- Overview of ObjectAgent
- Agent Networks
- Health Monitoring
- Error Handling
- Conclusion

Overview of ObjectAgent

ObjectAgent is an agent-based, message-passing software architecture that utilizes natural language processing to provide autonomous control to complex systems. Control systems are decomposed into agents, each of which is a multi-threaded process. Agents are used to implement all of the software functionality and communicate via a flexible messaging architecture. Each message has a content field written in natural language that is used to identify the purpose of the message and its contents. Agents may be loaded at any time and have the capability to configure themselves when launched, which simplifies the process of updating flight software and removes the complexity associated with software patches.

ObjectAgent agents are composed of skills. In Matlab, these skills are written as m-files with a specific format. Generally, each skill corresponds to one basic function, has inputs and outputs, and triggers one or more actions. These actions can include calling any other Matlab function. The skills that an agent possesses determine its complexity and functionality. However, all agents have a number of survival skills to ensure that they can communicate and recover from basic failures. Agent communication takes place solely through messages; there is no shared memory between agents. This ensures that agents can work together even when they are not located on the same processor.

ObjectAgent allows the user to specify the complexity of the agents and agent organizations and does not constrain users to a predefined notion of an agent. The user performs the decomposition of the system into agents. This allows greater flexibility, extensibility, upgradability, and compatibility with existing systems. An agent can dynamically accept new tasks and employ its skills to accomplish those tasks. Although artificial intelligence techniques are not built in to the ObjectAgent core, the OA system architecture allows AI techniques to be incorporated at any or all levels of the software. Many

tools are available to create skills. For example, the system includes fuzzy logic, neural net, system identification, learning control, access to Spacecraft Command Language, expert system and fault detection tools that the user can employ to solve his or her distributed control problems. These techniques can even be added after the system is in operation, which is not possible with today's flight systems. In addition, tools for enabling agent organizations are included. This permits agents to control the behavior of other agents.

Finally, special attention has been paid to developing a system that is easy-to-use and simplifies the flight software creation process. ObjectAgent is an integrated approach to agent and flight software design, making extensive use of simplified natural language and graphical user interfaces (GUIs). This design environment not only simplifies the agent creation process but also provides a common interface to a number of advanced control and estimation techniques.

The latest developments in the ObjectAgent architecture include GUIs to develop agent networks, specify health reports and organize a fault detection approach. Complementing the user interfaces tools is a background layer of communication and messaging which actually performs the health monitoring and error handling functions, based upon user-supplied information. This additional architecture enables health information to be distributed and errors to be handled in a flexible manner, and with minimal user intervention.

Agent Networks

An agent network is a group of agents that are connected to one another through various relationships. The behavior of an agent network is largely determined by the skills that they run, as well as the associated inputs and outputs of those skills. As discussed earlier, the sharing of information is established by conversations between agents, where the conversations lead to the creation of task lists. Nominally, agents can operate together in an environment without a priori knowledge of their relationships—they only know what they have, and what they need.

It is desirable in some cases, however, for the designer to establish specific relationships between agents. In these cases, there is an intentional effort to organize the network according to some distribution of responsibility that translates directly into the application for which the agents are designed. A graphical user interface has been developed for ObjectAgent which allows the rapid development of such networks. In this GUI, relationships between agents are easily added or removed, and the resulting network is displayed.

The purpose of creating a relationship is to quickly define a general protocol of interaction. Thus, when a specific relationship is defined between two agents, that carries with it an expectation of how those two agents will interact. ObjectAgent currently supports three types of agent relationships: master, slave, and peer.

Each of these relationships maps into various attributes. The following protocols are to be followed in ObjectAgent:

a) Both error reports and health reports are sent to masters and peers (others may be added)

b) Tasks may be assigned by masters, and requested by peers.

c) Changes to interfaces may be made by masters, and requested by peers.

Of these three protocols, only a) is currently enforced, though b) and c) are under development.

Error and health reports will be discussed in the following two sections. As explained earlier, agents operate by executing tasks. An agent may dynamically change its own task list, assign tasks to its slave agents, or request tasks of its peer agents. The interfaces of an agent dictate how it communicates with others. These too may changed dynamically, with masters having more authority than peers.

An agent network is created by assigning various relationships between 2 or more agents. The network is organized by assigning levels to each agent. Consider the

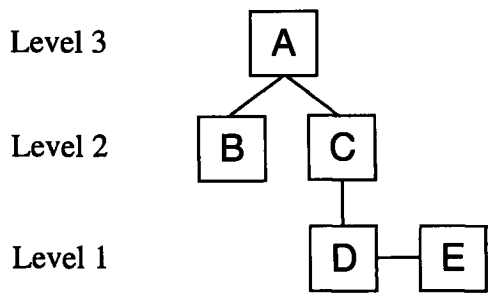

Figure 1: Example of an Agent Network

network in Figure 1, which is composed of agents A - E. In this diagram, masters are above slaves, and peers are next to one another.

In this network, agent A is a master to both B and C, agent C is a master to D, and D and E are peers. The level of each agent is dictated according to the following four rules:

1) If no relationships exist, the level is 1

2) In a master/slave relationship, the master level is 1 greater than the slave level

3) In a peer/peer relationship, the levels are equal

4) In any network, the lowest level is 1

It is easy to observe that all of these rules are satisfied in the example network. When the user attempts to create a relationship between two agents, the OA software first consults these rules. If the new relationship will not violate any of the four rules, it is allowed. Adding new relationships can affect the level of the other agents in the network. For example, let's say we wish to make agent E a master to agent F. The level of F will be 1 (rule 4), which means that E will have a level of 2 (rule 2). This of course will continue to impact the levels of the remaining agents in the network. The result is that agents A-E will increment their level by 1, thus satisfying all of the rules.

There are two incentives for establishing these level based rules. The first reason is that it helps to keep track of where each agent lies in the network. The higher the level, the more authority and access to information it has. The other reason is that it prevents users from establishing a poorly structured network. For example, let's say we wish to make agent E a master to agent A. This could result in commands circulating endlessly through the network, or other undesirable effects. The rules prevent such a set of relationships from being formed.

The primary objective in assigning relationships to agents is to provide an organizational structure which maps sensibly into the application for which they are designed. Once a network of agents is developed, a designer may press on with establishing the health monitoring and error handling characteristics for their system.

Health Monitoring

One of the most important tasks involved with controlling a complex system is to provide information about the system's health. Satellites are an excellent example—several important parameters, such as temperature, fuel, attitude, and power, are constantly being stored in memory and telemetered down to earth. In an agent-based system, where the computational tasks are distributed, it is especially important to monitor health. In ObjectAgent, a health monitoring architecture has been established which accomplishes two goals. First, it enables the user to easily define how agents conduct health monitoring. Second, it carries out health monitoring in the background for an agent network of any size.

Health monitoring is conducted automatically for a network of agents that are defined through master/slave or peer/peer relationships. Each agent has its own health report, which is defined by the user. A health report consists of three types of information: an overall *health number*, a set of *important parameters*, and a descriptive *list of errors*.

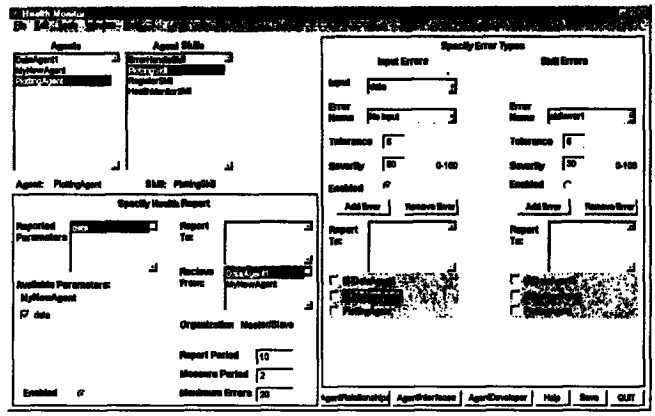

Figure 2: HealthMonitor GUI for ObjectAgent

Figure 2 shows the HealthMonitor GUI in Matlab, where the user specifies the details of each agent's health report. By default, health reports are automatically sent to master and peer agents, although the user may define additional destinations. Other decisions include the health measurement period, the report period, the maximum number of errors to include in any report (for agent memory purposes), and whether *health monitoring* is turned on or off.

The health number of an agent is an instantaneous measure of "how well" the agent is doing. The current method of measuring the health number is to begin with a nominal value of 100. If the agent experiences an error, its health is reduced by the severity associated with that error (see the Error Handling section). When the agent recovers, its health is restored by the same amount.

A more detailed set of information is included in the list of important parameters. All of the outputs of skills in a particular agent are available to be selected as important parameters. They are measured at a specified rate, and their name, value, and time of measurement are included in the health report.

The final element of the health report is a list of errors. Each time an agent detects an error, a packet of relevant information is stored in the agent data structure (this is discussed further in the Error Handling section). When it is time to send the next health report, any new error information is included.

When health monitoring is "enabled" in the GUI, the *HealthMonitorSkill* is added to the agent. As discussed earlier, the Matlab representation of a skill is a script m-file. Skills are updated according to their update period, which, for the HealthMonitorSkill, can be different for each agent. The update period for this skill is automatically set to the minimum of what the user selects

for the measure and report periods. When the skill updates periodically, it does so to either measure or report health. The measurement is performed by taking a snapshot of the agent's health number and important parameters at that instant in time, and concatenating that information to the current health report. All health information is stored in the agent until it is sent, at which time the memory is cleared.

When it is time to send the health report, the HealthMonitorSkill adds the task, "*send message*" to the agent's task list. Here, *message* is a parsed data structure that contains the health report itself, the name of the receiving agent, as well as instructions for the receiving agent to update its own HealthMonitorSkill. This automatic update is done because the user may desire to carry out some specific actions each time a health report is received, such as print the health report to a file. They may code any such actions into the user-defined section of HealthMonitorSkill.

Once a network of agents is established, a designer may simply use the HealthMonitor GUI to design their method of monitoring health among agents. It should be noted that both external parameters (i.e., temperature measurements) and internal parameters (i.e., time to run an algorithm) may be monitored. What the designer decides to do with this information is, of course, case specific.

Note that a master agent that has no peers or masters to itself will only *receive* health reports from other agents. Thus, it would be sensible to allocate a large block of memory to a high level agent that would be responsible for receiving, analyzing, and storing health information. Similarly, artificial intelligence techniques could be added in at any level of the network to react to the system health.

Error Handling

The fault detection architecture for ObjectAgent was designed with two objectives in mind: 1) to provide a flexible framework to detect, report, and recover from errors in a distributed agent environment; 2) to minimize the amount of required user-intervention. The user is required to supply the necessary algorithms for detection and recovery, and to make selections in the HealthMonitor GUI. The ObjectAgent software then uses this information, along with the ErrorHandleSkill and appropriate FDIR functions, to implement the error handling.

The main element of the fault detection architecture is an error. In ObjectAgent, an error is defined as follows:

> *A specific occurrence, with a unique name, that may be detected and recovered from in a distinct manner.*

There are 3 primary actions surrounding the occurrence of an error: detection, recovery, and reporting. In an agent-based framework, once an error is detected, that knowledge is initially isolated to the original agent. The recovery may be performed by the original agent or by an outside agent, providing it has been informed. ObjectAgent uses its message passing architecture to distribute error information to all appropriate agents, thereby allowing a distributed recovery approach to be used.

The user-supplied detection and recovery algorithms are contained in FDIR functions. In Matlab, these functions are script m-files that share the name of the associated error. Once a user identifies an error for their system and assigns it a name, they can introduce the error into the OA environment by using the HealthMonitor GUI.

Referring to Figure 2, the right side of the GUI is used for specifying the attributes of errors. The types of errors which may occur are divided into 2 categories: *Input Errors* and *Skill Errors*. The error handling section of the GUI is split according to these types. For either type of error, the GUI allows the user to do the following:

- Add or Remove an error
- Set the Tolerance
- Set the Severity
- Select which agents the error is reported to

An Input Error is any error that has to do with a specific input to a skill, while a Skill Error is an error that occurs while a skill updates. For example, "No Input" is a generic Input Error that occurs when an input is expected, but is not received. The default action is to seek for a new source. The detection and recovery algorithms for this error have been written into an FDIR function, which may be applied to any input of any skill in any agent of the system.

Errors may be either environmental or software oriented in nature. An error such as "Bad Signal" or "High Temp" would be environmental, whereas "No Input" is software oriented because it refers to a skill not receiving an expected input from another software agent.

A tolerance is associated with each error in order to give the agent sufficient time to recover. By providing a large enough tolerance, the agent has enough time to complete its recovery before starting the process again. The severity is a user-defined parameter which is related to the health. As discussed earlier, the occurrence of an error causes the health to be reduced by that error's severity.

Finally, the user may select which agents are informed when an error occurs. Upon detection, a description of the error is immediately reported to any master or peer agents that the agent may have. The GUI allows users to add other agents to this list. It is useful to report errors to other agents for two reasons. First, other agents are likely to be affected. If they are informed, then they have a chance to

minimize the negative impact that the error may have on them. Second, it may be useful for other agents to assist in correcting the error. For example, in a master/slave relationship, it may be desirable for the master to be the sole decision-maker. When the slave experiences an error, it reports to the master, and the master performs the recovery algorithm, which would include new instructions or tasks for the slave.

The following example, which is used in the ObjectAgent Tutorial, demonstrates the error handling features of ObjectAgent. A simple network of 3 agents is simulated. The structure is illustrated in Figure 3.

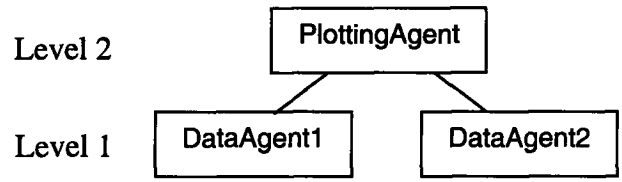

Figure 3: Simulated Network

DataAgent1 and DataAgent2 are identical; each possesses the skill "DataSkill", which simply calculates the system time and sends it as an output called "data". PlottingAgent has the skill "PlottingSkill", which plots the input "data" as a function of time. Prior to running the simulation, DataAgent1 is defined as PlottingAgent's source of "data".

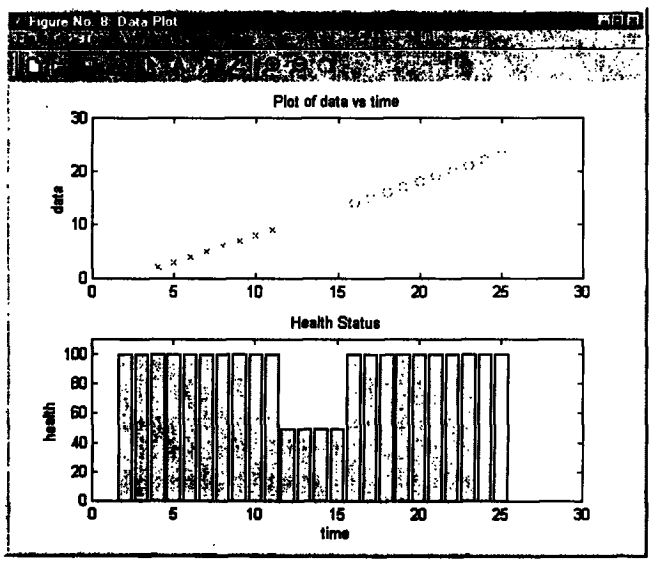

Figure 4: Health and "data" of PlottingAgent when "No Input" error occurs

The results of the simulation are shown in Figure 4. The goal is to illustrate that the error handling architecture works properly by using the "No Input" FDIR function. If the "No Input" error is detected by PlottingAgent, it should seek for a new source, and find that source in DataAgent2. PlottingSkill is coded such that, when the source of "data" changes, it plots "o" instead of "x".

At 10 seconds DataAgent1 is failed, preventing PlottingAgent from receiving "data". PlottingAgent detects the "No Input" error at 12 seconds (due to a 2 second delay), and reduces its health by the severity of the error, which was set to 50. After 4 seconds pass, the recovery algorithm is shown to be successful when "data" is received from DataAgent2.

Although this is an extremely simple example, it illustrates the ease with which error handling may be executed. The detection and recovery scheme of this particular error may be applied to any input of any skill of any agent.

Conclusion

In summary, an architecture has been created for ObjectAgent which facilitates both health monitoring and error handling for an agent network of any size. Graphical user interfaces enable users to easily establish relationships in a multi-agent system, edit an agent's health monitoring characteristics, and assign error handling capability to agents in a modular fashion. In this architecture, error and health information is made readily available throughout the agent network, extending the potential for robust autonomy.

References

Schetter, T. P., M. E. Campbell, and D. M. Surka. 2000a. Comparison of Multiple Agent-based Organizations for Satellite Constellations. In Proceedings of FLAIRS 2000. Orlando, Florida.

Schetter, T. P., M. E. Campbell, and D. M. Surka. 2000b. Multiple Agent-Based Autonomy for Satellite Constellations. In Proceedings of the Second International Symposium on Agent Systems and Applications. Zurich, Switzerland.

Surka, D. M., M. C. Brito, and C. G. Harvey. 2001. Development of the Real-Time ObjectAgent Flight Software Architecture for Distributed Satellite Systems. To be Presented at IEEE Aerospace Conference 2001. Big Sky, Montana.

Zetocha, P., L. Self, R. Wainwright, R. Burns, M. Brito, and D. Surka. 2000. Command and Control of a Cluster of Satellites. *IEEE Intelligent Systems.*

Zetocha, P. and M. C. Brito. 2001. Development of a Testbed for Distributed Satellite Command and Control. To be Presented at IEEE Aerospace Conference 2001. Big Sky, Montana.