The Real-Time ObjectAgent Software Architecture for Distributed Satellite Systems[1]

Derek M. Surka
Princeton Satellite Systems
150 S. Washington St. Suite 201
Falls Church, VA 22046
703-237-8484
dmsurka@psatellite.com

Margarita C. Brito and Christopher G. Harvey
Princeton Satellite Systems
33 Witherspoon St.
Princeton, NJ 08542
609-279-9606
{megui, charvey}@psatellite.com

Abstract—The ObjectAgent system is being developed to create an agent-based software architecture for autonomous distributed systems. Agents are used to implement all of the software functionality and communicate through simplified natural language messages. Decision-making and fault detection and recovery capabilities are built-in at all levels.

During the first phase of development, ObjectAgent was prototyped in Matlab. A complete, GUI-based environment was developed for the creation, simulation, and analysis of multi-agent, multi-satellite systems. Collision avoidance and reconfiguration simulations were performed for a cluster of four satellites.

ObjectAgent is now being ported to C++ for demonstration on a real-time, distributed testbed and deployment on TechSat 21 in 2003. The present architecture runs on a PowerPC 750 running Enea's OSE operating system. A preliminary demonstration of using ObjectAgent to perform a cluster reconfiguration of three satellites was performed in November 2000.

TABLE OF CONTENTS

1. INTRODUCTION

There is an increasing desire in many organizations, including NASA and the Department of Defense, to use constellations or fleets of autonomous spacecraft working together to accomplish complex mission objectives. Some of the many advantages of using distributed satellite systems include greater performance, lower cost, and improved fault tolerance, reconfigurability and upgradability. Coordinating the activities of all the satellites in a constellation is not a trivial task, however, and the use of software agents for this task is a promising technology.

Princeton Satellite Systems is developing the ObjectAgent (OA) and TeamAgent systems under Air Force Research Laboratory (AFRL) Phase II Small Business Innovative Research (SBIR) funding to create an agent-based software architecture that is designed for autonomous, distributed systems. Agents are used to implement all of the software functionality and communicate through simplified natural language messages. Decision-making and fault detection and recovery capabilities are also built-in at all levels.

The TeamAgent system applies ObjectAgent to the problem of controlling multiple cooperative satellites. TeamAgent enables agent-based multi-satellite systems to fulfill complex mission objectives by autonomously making high- and low-level decisions based on the information available to any and/or all agents in the satellite system. Simulations of multi-agent systems for multiple satellites have been developed using TeamAgent to illustrate collision avoidance and reconfiguration for a four-satellite constellation. Agents were used to monitor for collisions, reconfigure the fleet, and optimize fuel usage across the cluster during reconfiguration.

[1] 0-7803-6599-2/01/$10.00 © 2001 IEEE

Previous papers have addressed the Matlab prototyping of ObjectAgent/TeamAgent and have described the research into agent organizations for distributed satellite control [7] [8]. These papers have also described the various multi-agent, multi-satellite simulations that have been assembled.

This paper describes the status of the real-time C++ development of ObjectAgent/TeamAgent. The first section describes the motivation for using software agents to control distributed satellite systems. It also describes the philosophy behind the development of ObjectAgent. The second section provides an overview of the agent architecture as implemented in Matlab. The third section describes the C++ implementation of this architecture and compares it to the Matlab prototype. Particular emphasis is placed on the message processing and the multi-threaded nature of the C++ agents. The effects of real-time operating system selection are also discussed. The fourth section describes a simple demonstration of real-time ObjectAgent. Finally, the current status of the ObjectAgent port from Matlab to C++ is presented and directions for future work are provided.

2. AGENTS FOR SPACECRAFT AUTONOMY

The use of software agents is becoming increasingly popular as a method to improve the level of spacecraft autonomy. There is no consensus on the exact definition of a software agent, but a standard definition is given by Weiss [9]:

> *An agent is a computational entity that can be viewed as perceiving and acting upon its environment and that is autonomous in that its behavior at least partially depends on its own experience.*

Bradshaw [3] and Knapik and Johnson [6] provide good overviews of the many different definitions used by researchers in the fields of artificial intelligence (AI) and computer science.

The major benefit of agents is their autonomy. Intelligent agents can be given high level goals and then autonomously determine the appropriate actions to fulfill these goals. This can include interaction and collaboration with other agents. Agent based software is a form of distributed programming and as such, it maps naturally onto the requirements of distributed spacecraft [7]. Part of the purpose of the development of ObjectAgent is to research and demonstrate those areas where agent-based software can benefit distributed satellite systems.

For example, it is easy to imagine each spacecraft having a higher-level agent that coordinates its activities with those of other spacecraft. This is important for those proposed missions that require multiple cooperative satellites to achieve their objectives, such as space-based interferometery. The agents and not the ground operators will be responsible for activity coordination.

Additionally, cluster-level agents will enable operators to command the entire cluster as a "virtual" satellite by decomposing high-level commands or goals into individual spacecraft commands that can be sent to the appropriate spacecraft-level agents. This can be done without the operator having detailed information about the state of each spacecraft. Although the use of software agents may not be the only method to enable a fleet of spacecraft to work together, it appears to be very promising, especially for larger clusters of satellites.

In addition to enabling more complex cooperation among satellites, there are many potential benefits to using agents onboard individual spacecraft. These advantages are analogous to those for clusters since a similar decomposition of intelligence and software functionality into hierarchical agents can be performed on each spacecraft. These benefits include:

- Greatly increasing the level of autonomy onboard the spacecraft by flowing down high level tasks;
- Making flight software flexible and easy to adapt because the agents can be dynamically loaded;
- Improving the reliability of spacecraft and fleets of spacecraft by incorporating fault detection at both high and low levels; and
- Reducing the need for large ground support organizations.

NASA's Deep Space 1 Remote Agent Experiment

The first demonstration of agents used for control onboard spacecraft was NASA's Deep Space 1 (DS-1) Remote Agent Experiment [1]. During the experiment in May 1999, the spacecraft was sent a list of goals instead of the usual detailed sequence of commands to execute. The Remote Agent (RA) software generated a plan to accomplish these goals and then executed this plan, monitoring for hardware faults during execution. Despite some minor glitches, the Remote Agent Experiment was a complete success and achieved 100% of its validation goals [2].

The RA software is installed as a layer on top of the regular flight software, an approach that requires the agent to process a lot of information and to be very intelligent. The complexity of using this approach to space flight software was evident when most of its capabilities were stripped off prior to launch and replaced by the more conventional Mars Pathfinder software [5]. (Additional capability was uploaded after launch.)

ObjectAgent Software for Autonomous Spacecraft

The ObjectAgent Software Architecture goes beyond the DS-1 remote agent experiment by using agents as the basis of the system rather than as a top layer. This is a key feature that distinguishes ObjectAgent from other agent

architectures. Each agent is a multi-threaded process and this architecture allows decision-making, including fault detection and recovery capabilities, to be built in at all levels of the software. This in turn alleviates the need for extremely intelligent high-level agents and simplifies the software interfaces.

A fundamental component of ObjectAgent is the flexible messaging architecture that provides a reliable method for agent-to-agent communication both on a single processor and across networks. Each message has a content field written in natural language that is used to identify the purpose of the message and its contents. Natural language was selected so that users could easily send messages or commands to the agents as well as understand the messages being sent between agents. The latter is important for debugging purposes. Because all agent-to-agent communication is through messages, it does not matter where the agent is located. Thus, this architecture can encompass multiple processors on one spacecraft or multiple processors on different spacecraft and on the ground.

Two additional advantages of using agents for all software functionality are increased flexibility and robustness. Robustness is improved in ObjectAgent because all agents are endowed with a set of basic survival skills. Each agent has knowledge of its skills, inputs, and outputs, and is capable of automatically configuring itself upon launch. It will automatically seek out other agents who can provide it with the inputs it needs as well as other agents who need its outputs. In this sense, an ObjectAgent system is self-organizing.

These same survival skills enable agents to be dynamically added to a system to improve the system capabilities or recover from a failure. The flexible and reconfigurable messaging architecture provides a common software interface that is vital to this ability to dynamically add or change software. Since all software is implemented as agents with the common messaging interface, all software can be easily replaced or updated. This messaging architecture also helps the system to recover from failures.

Another key feature of ObjectAgent is it 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. If the user desires a system that runs traditional flight software, this software can be encapsulated in a single agent. This agent would then be the only piece of software running on the flight computer. If the user wishes to make use of the advantages of an agent-based system, however, she can distribute the traditional flight software functionality among multiple agents in any fashion she desires.

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. These techniques can even be added after the system is in operation, which is not possible with today's flight systems.

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.

The following sections provide more detailed information about how this ObjectAgent architecture is implemented in both Matlab and C++.

3. OBJECTAGENT MATLAB ARCHITECTURE

ObjectAgent was first prototyped in Matlab for a number of reasons that benefit both OA architecture developers and end-users.

For the OA architecture developers, the use of Matlab enabled the proposed agent architecture to be rapidly prototyped and tested. An integrated development environment (IDE) for ObjectAgent was easily created using the Matlab GUI functions and this design environment is relatively platform independent, limited only by the platforms that Matlab supports.

OA end-users can quickly verify their agent and algorithm designs as well as take advantage of the wide variety of Matlab-based software and toolboxes in existence, including PSS' Spacecraft Control Toolbox. In the future, OA users will be able to automatically convert their Matlab agents into C++ agents that can be deployed in real-time systems.

There are however, some limitations to using Matlab for the simulation of agents. Matlab is a single-threaded application so agents cannot be multi-threaded. Message passing must be emulated since Matlab does not have built-in support for messaging. These limitations preclude the use of Matlab for multi-agent system performance verification.

Despite these limitations, the use of Matlab for rapid prototyping and platform independence has made it a very beneficial tool for the development of ObjectAgent and multi-agent systems.

Matlab Agent Architecture

Matlab agents are composed of skills that are written as Matlab m-files. Generally, each skill corresponds to one basic function, has inputs and outputs, and triggers one or more actions. There is a specific format for these skill files and each skill contains a data structure field that describes

the assigned priority, the update period, the input and output interfaces, and the communication method.

The primary action for each skill is an update action that is run every time the agent invokes the skill. This update action can make use of any Matlab function or m-file and dictates the skill functionality. Typically, the update function will process the skill inputs and generate the appropriate outputs that will be used by other skills within the same agent or sent to other agents through messages.

There is no limit to the number of skills an agent can possess and there is a set of common skills required by each agent. One such skill is the RegisterSkill. Each agent must register with its home "Message Center" before it can send or receive messages or perform any task processing. The next two subsections describe this messaging architecture and task processing in greater detail.

Messaging Architecture

This section describes in greater detail the format of the messages used in the Matlab version of ObjectAgent for all agent communication. Note that this is also the format for all agent tasks (described in the next subsection.) The message format presently in use is proprietary and uses simplified natural language.

The important characteristics of the messages used in ObjectAgent are:

- Each message has a content. The content tells the agent what the message is about. It is a string composed of a subset of natural language structured in one of two formats:

  1. verb / noun phrase / preposition / noun phrase
     e.g. "receive data for Agent(Skill)"

  2. verb / noun phrase / preposition / noun phrase / preposition / noun phrase
     e.g. 'find source for data for Agent(Skill)'

  There are presently 14 verbs understood by the agents and users can expand this list.

- Each message has data associated with it. The data can be anything that one agent wants to send to another. For example, this could be the parameter values that are being sent to another agent (such as in the first example above) or it could be the name of the desired information (such as in the second example.)

- Each message is time-tagged. This makes it easy for agents to determine if the data is valid.

- Each message has fields indicating to and from whom the message is being sent.

- Each message has a priority associated with it. Higher priority messages are processed first.

This message format can easily be adapted to other industry formats.

As mentioned previously, Matlab does not provide built-in support for message passing so the messaging between agents must be emulated. This is achieved by the creation of a "message center" that controls the flow of messages between agents. Every agent must register with its home message center before it can send or receive messages or perform any task processing. The message center is responsible for routing messages between agents.

Figure 1 shows how this message passing architecture is implemented in Matlab. A collision avoidance agent on one spacecraft is sending a MoveCollAvoid message (MoveCollAvoid is a user-defined verb used in [7] and [8]) to an orbit maneuver agent on another spacecraft. This message is routed by the message center on the first spacecraft (MC1) to the message center on the second spacecraft (MC2) and then on to the appropriate agent.

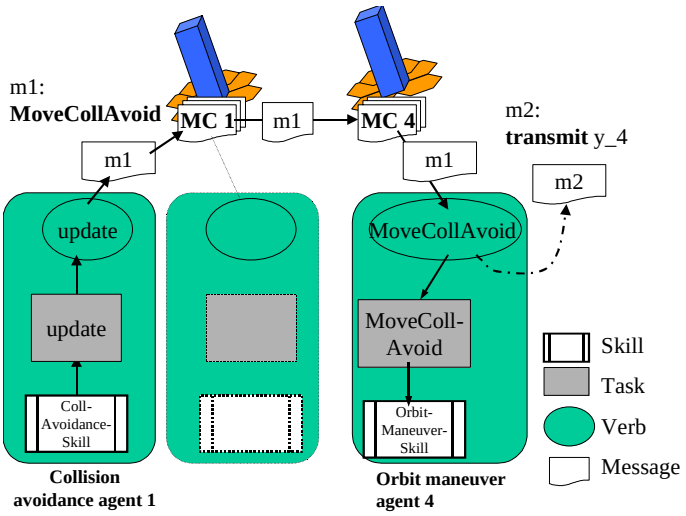

Figure 1. Example of the Matlab Messaging Architecture

Agent Task Processing

Matlab agents are single-threaded and their actions are based on the processing of their task lists at every time step. Tasks and messages have the same data structure, making for a fast and clean implementation and enabling tasks to be sent to the agents in messages. Each agent dynamically maintains its own task list and at every time step, the entire task list is processed in the manner shown in Figure 2.

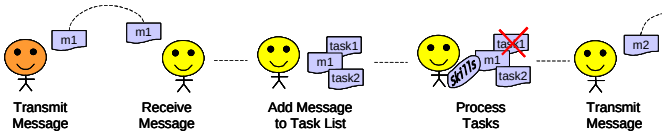

Figure 2. Agent Task Processing

First, the agent's incoming messages are added to its task list. This new task list is then prioritized and the tasks/messages with highest priority are processed first. The tasks are processed through natural language processing in which the verb determines the actions taken by the agent. These tasks, when processed, can cause a message to be created and sent, and/or actions to be taken by the agent that changes its internal state.

Figure 1 shows an example from [8] where the task "update CollAvoidSkill" creates the message "MoveCollAvoid sc_4" (m1) and sends it to the orbit maneuver agent because a possible collision involving spacecraft #4 was detected. The verb function MoveCollAvoid is then processed by the orbit maneuver agent, which causes the update action of its OrbitManeuverSkill to be run. Additionally, message m2 is transmitted back to the collision avoidance agent 1.

Creating Matlab Agents

The ObjectAgent Matlab design environment uses an integrated graphical user interface to speed development. Figure 3 shows the primary interface. The user can design and simulate agents from this window.

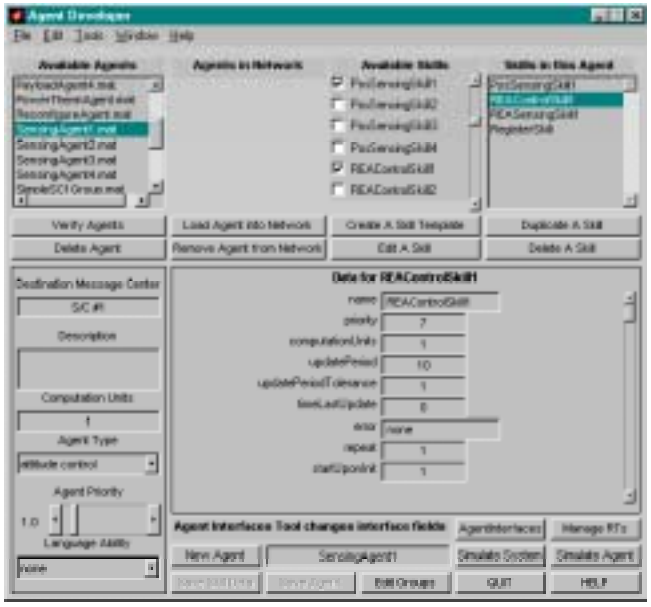

Figure 3. Agent Developer GUI

Agent skills are available in a library database making it easy to reuse code. Skills are added to an agent by selecting them from a checkbox list in the top right-center of the window. Skill parameter values can be changed from their default values in the bottom half of the window.

Similar GUIs are available to specify agent communications, relationships, and error reporting.

4. Real-Time C++ ObjectAgent Architecture

The overall objective of our research is to use ObjectAgent to control real-time, distributed systems. This requires that the OA architecture be ported to a real-time programming language. C++ was selected because its object-oriented nature meshes nicely with the OA design philosophy. It was selected over Java and other object-oriented languages because it is more commonly used for the control of real-time systems and because of our greater experience with C++.

After selecting the programming language, it was necessary to select a target real-time operating system (RTOS) and embedded processor. The current baseline processor is the PowerPC 750 and the current architecture is designed to run on top of Enea's OSE operating system. The PowerPC 750 was selected because of its high speed and memory capabilities and because a rad-hard version will soon be available from Lockheed-Martin.

RTOS Selection

Enea's OSE was selected as the operating system for ObjectAgent for several reasons. OSE is a message based operating system designed for distributed architectures with many features that lend itself to the ObjectAgent architecture. These features include multi-threading, very good process and memory management, and dynamic process loading, and are discussed in greater detail below.

Message based RTOS simplifies the design of distributed applications by enabling designers to implement their applications using high-level constructs such as state transitions and message passing. They typically provide built-in safety features that make it ideal for distributed applications that require high availability and security.

Unlike traditional embedded operating systems, which utilize lightweight tasks to partition complex activity and semaphores to establish communications, the messaging RTOS use memory-protected processes and message-based communications. This approach makes it easier to conceptualize complex applications and distribute programming responsibilities across large development teams. The messaging RTOS model also makes it easier to compartmentalize critical operations and data, thereby enhancing reliability and security.

The following subsections describe how OSE differentiates itself from traditional kernels like VxWorks. We selected OSE as our initial target RTOS because these features better support distributed systems and simplify the work that must be done by our developers.

Process Management—Processes may be grouped into blocks, each with its own memory pool. While other kernels

may schedule tasks running in a shared memory environment, OSE knows what resources each owns, including such things as file descriptors, sockets, as well as all memory resources, and supervises to avoid conflicts. If tasks die, the kernel can reclaim the resources automatically. Additional memory features are discussed in the Memory Management Unit (MMU) support section.

Interprocess Communication—OSE is a true message passing OS which features full central processor unit (CPU) or destination transparent messaging. In this sense, it is designed for distributed processing applications. The messaging schema naturally supports fault tolerant and/or high availability designs, in the following fashion:

- Processes send messages to other processes
- Processes dynamically bind to other processes
- OSE supervises all communications between processes; if delivery fails, or a process dies (or is killed), all connected processes are notified so they may take corrective action. One corrective action could be to establish a connection with a backup process (or board) or messages may be dynamically re-routed to alternate destinations.

Error Handling—Error handling is managed by the process, block, or system error handlers. This simplifies code development, and offers a hierarchical approach to error management. This will facilitate agent error handling.

MMU Support—Integrated MMU support means, among other things, processes or blocks (groups of processes) can be partitioned into MMU segments. Depending on the level of security desired, messages passed between processes could either be copied between segments or accessed by reference. The latter will be important when large amounts of data must be manipulated. This would reduce the need for shared memory pools.

Dynamic Software Upgrades—Software can be dynamically upgraded even to the individual process level, without stopping the rest of the system. This is enabled by the fact that processes bind (and detach) from one another dynamically. This feature is important for adding and removing agents dynamically.

Agent Architecture

Each C++ agent is an object of a specific Agent subclass. A generic C++ Agent class has been created and all Agent subclasses inherit from this class. The generic Agent class contains a number of the various survival skills required by each agent, such as the abilities to communicate through messages and to seek for input sources. The Agent class (and all of its subclasses) contains an Update function in which all processing for the main agent process takes place.

C++ agents are composed of skills that are written as C++ classes. Generally, each skill acts in the same fashion as those skills found in the Matlab agent architecture. Each skill class contains data fields that describe the assigned priority, the update period, the input and output interfaces, and the communication method.

The primary action for each skill is an update function that is executed every time any agent invokes the skill. This update function can make use of any C++ function or class and dictates the skill functionality. The update function operates in the same fashion as the update function described in the Matlab agent architecture.

The skills of an individual agent can be grouped into activities. Each activity runs in a separate thread (or OSE process) and the threads of one agent share common memory. The activities are defined in the specific Agent subclasses.

There is no memory sharing between agents. All synchronization of agents is by messaging. Within an agent, other mechanisms may be used.

OSE Agent Implementation

Figure 4 shows the C++ implementation of an agent in OSE. An agent consists of three primary OSE processes — the Main Agent process, the Dispatcher process, and the Collection Center process — and any number of additional skill or activity processes.

The Main Agent process is the first process to run when an agent is initialized. This process is responsible for creating the common objects required by the agent and for starting the other processes. Upon initialization, the Main process first creates an instance of a specific Agent subclass. The Main process then creates numerous instances of a Queue class that serve as inboxes and outboxes for agent communications. These queues store Message objects and are discussed in greater detail in the next subsection.

After the creation of the message queues, the Skill objects used by the agent are created. This is followed by the creation of a table mapping input names to the appropriate skills. The Collection Center and Dispatcher processes are then created and are passed pointers to the appropriate queues. Inboxes are used by the Collection Center to pass incoming messages to the appropriate processes. The outboxes contain messages that are generated by the agent and that are to be sent out by the Dispatch process.

Finally, any other skill or activity processes are created. Each skill process has an inbox and an outbox associated with it. After initialization, the Main Agent process continues to run and repeatedly calls the agent object's update function. The Main Agent process is a prioritized process.

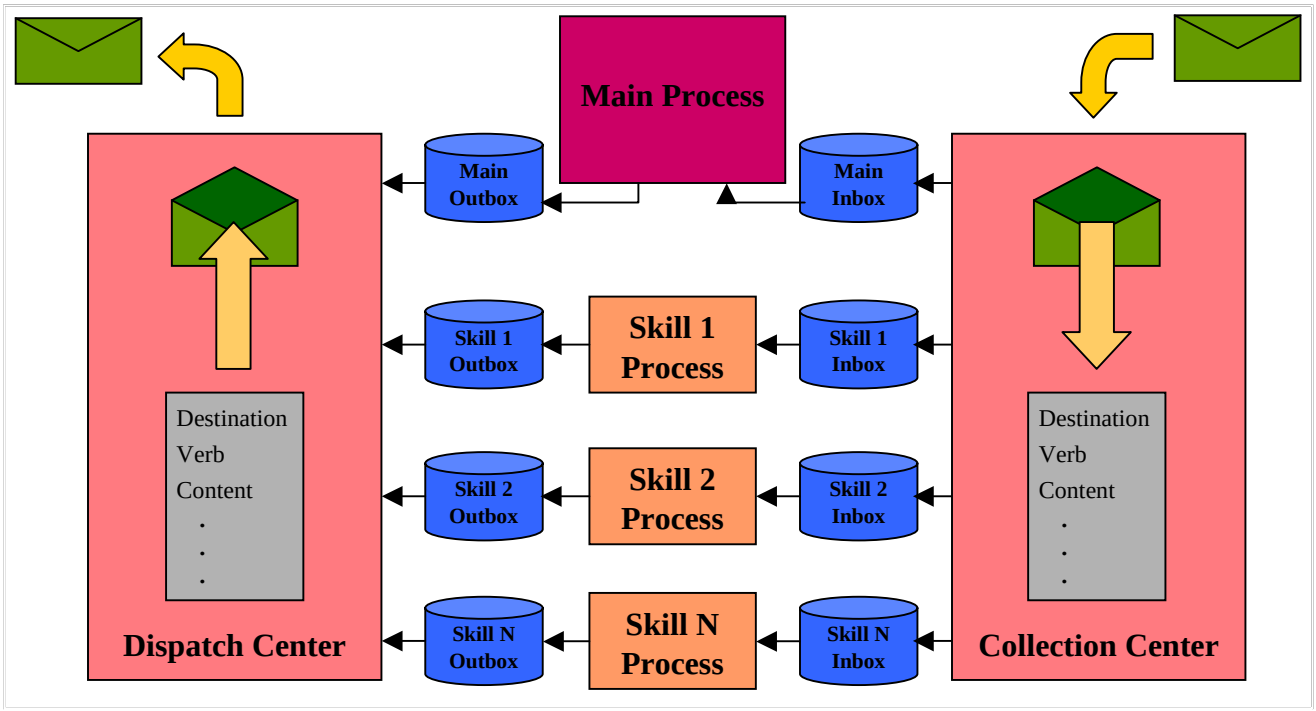

Figure 4. C++ Agent Architecture

Messaging Architecture

OSE provides built-in support for message passing through the use of signals. Signals are passed from one OSE process to another and the signal types are user-definable. A C++ Message class has been defined for agent messaging. The important characteristics of this Message class are the same as those described in the Matlab agent architecture although some modifications have been made to the implementation.

The first modification is that messages are no longer processed based on priority. They are processed on a first in, first out (FIFO) basis, which is partly due to the way OSE handles signals. The second modification is that there is no broadcast capability at present, again because of limitations with OSE. We will implement some form of broadcasting ourselves.

Two additional fields have also been added to the class to identify the verb and the type of data being sent. Since all message data is sent as a character string, it is necessary that the receiving agent know the type of data being sent in order to be able to reconstruct it. The verb has been stripped out of the message content and placed in a separate enumerated field to reduce the need for an agent to use natural language processing (NLP) to understand each message. Simple messages such as "request" or "transmit" will not require NLP although more complex messages will still make use of this function. The user will still interface with the agents through natural language by virtue of a pre-processor that will convert Matlab-type OA messages into C++ objects and vice versa. We had contemplated using a different OSE signal type for each verb but that would have made the architecture too OS-specific.

Figure 4 also shows the messaging architecture as implemented in C++. Each agent creates Dispatcher and Collection Center processes that handle outgoing and incoming messages, respectively. When these processes are created, pointers to either the agent's inboxes or outboxes are sent to each. The queues are used as shared memory between the agent processes and its Dispatcher and Collection Center.

To send a message, an agent or skill process first creates a Message object and places this object in its outbox to be sent by the Dispatcher. Dispatcher processes always send their messages to another agent's Collection Center process. Since each message contains a destination, the Dispatcher is able to find the destination Agent's Collection Center process and send the outgoing message (via an OSE signal) to it.

Unfortunately, message objects cannot be passed within an OSE signal. Therefore, when a Message object is generated by an Agent to be dispatched, it first must be decomposed by the Dispatcher into simple data types such as character buffers, integers, and floating point values. These decomposed values can then be stored and sent within OSE signals.

When this type of signal is received by a Collection Center process, a new message object is created by the Collection Center with the appropriate data fields. Based upon the verb

and content, this message is then stored in the inbox queue of the appropriate agent process. If the verb is neither "request" nor "transmit", the message is passed to the Main Agent for further processing. Otherwise, the Collection Center compares the name of the input/output (contained in the message content) with the table mapping inputs and outputs to skills to determine the inbox(es) into which the message should be placed. The agent and skill processes poll their inboxes periodically, dequeue incoming messages, and execute the code indicated by the message object.

This architecture enables all natural language processing to be localized within the Main Agent process and invoked only when necessary. For the simple demonstrations presented to date, NLP has not been required.

Agent Processing

Unlike the Matlab architecture in which an agent is single-threaded and processes a task list every time step, C++ agents are multi-threaded and their actions are based on priority within the operating system.

Agents are designed to carry out certain activities. These activities are implemented as OSE processes and spawned by agents. Activity processes accomplish their goal by using one or more instances of particular skills. Skill subclasses inherit from a generic skill class, which also has an Update function in which all processing is accomplished.

Activities invoke their Skill objects' Update functions in the appropriate order to accomplish their final objective. An activity process can be implemented so that once processing is complete, it halts and restarts from the beginning when signaled to restart.

Another possible implementation of an Activity process is to allow it to run continuously in parallel with all other processes. These Activities can be stopped and started when necessary.

Issues that are still being addressed include the prioritization and coordination of the multiple agent processes. Deadlock may be addressed through the use of some of the basic survival skills discussed earlier. Agents are designed to time-out when they do not receive information from other agents in a timely fashion and to either report this condition or take corrective action internally, if possible. Watchdog agents will be created to monitor agent CPU and memory usage and to shut down those agents that are not behaving properly. These issues will be addressed in greater detail throughout the next year.

Creating C++ Agents

Currently, C++ agents are created by hand. Every agent class inherits from a base Agent class, allowing the user to focus on adding the desired features to the new agent without having to redefine the basic agent functionality.

The use of similar Matlab and C++ function libraries simplifies the conversion of Matlab agents to C++. In the future, a GUI will be created to perform this conversion automatically given user preferences.

5. Simple Real-Time OA Demonstration

The first significant demonstration of the real-time ObjectAgent architecture took place in November 2000. During the demonstration, ObjectAgent/TeamAgent was used to control the reconfiguration of a cluster of three satellites.

Demonstration Scenario

The real-time demonstration scenario is a variant of one of the reconfiguration scenarios performed initially by Schetter [7] and [8] in the Matlab/Simulink environment. This scenario was later modified for implementation in the pure ObjectAgent Matlab design and simulation environment.

One cluster of three satellites was simulated. The reference orbit was a circular orbit with a semi-major axis of 7100 km and an inclination of 28 degrees. The three spacecraft were placed in an elliptical trajectory relative to the reference orbit. The orbits are described by the force free solution of Hill's equations relative to the reference orbit. The free elliptical solution was chosen such that:

1. The free elliptical trajectory traced out a 2:1 ellipse in the vertical plane of motion (x-z plane where the coordinate frame used is the local-vertical, local-horizontal (LVLH) frame of the reference orbit — x is in the direction of the velocity vector, z is nadir pointing, and y completes the right-handed coordinate system);
2. The projection of the free elliptical trajectory was a circle in the local horizontal plane (x-y plane) of radius 5 km; and
3. The free elliptical trajectory was inclined ±26.57° from the local horizontal plane (x-y plane.)

The spacecraft were equally distributed about the ellipse. The initial orbital elements of the three spacecraft are given in Table 1.

Table 1. Demonstration Orbital Elements

	S/C #1	S/C #2	S/C #3
Semi-Major Axis (km)	7100.003	7100.007	7100.007
Eccentricity	3.520e-4	3.524e-4	3.524e-4
Inclination (degs)	28.000	28.035	27.965
Longitude of Ascending Node (degs)	-0.09	0.04	0.04
Longitude of Perigee (degs)	-179.92	-59.92	59.84
Mean Anomaly (degs)	180.00	59.88	-59.88

One spacecraft was selected as the Cluster Manager and was in charge of maintaining the proper spacecraft formation. Upon receipt of a ground command to change the radius of the projected formation circle, the Cluster Manager decided where each spacecraft should move to by optimizing fuel usage across the cluster.

These new positions were sent to each spacecraft and they would then plan a series of burns to move themselves to those locations.

Testbed Environment

The real-time testbed that was used for the development of the demonstration is shown in Figure 5.

Figure 5. ObjectAgent Real-Time Testbed

The chassis on the upper-left side of the figure houses the three PowerPC 750 boards. Each board represents one of the three spacecraft and runs OSE and the real-time ObjectAgent software. All communication among boards and the simulation and development computers uses Ethernet.

The simulation of the three spacecraft resides on the PowerMac G4, located in the lower right of the photo. The simulation is written in C++ and simulates the attitude, orbit, and hardware dynamics of each spacecraft as well as the space environment. The boards "sense" and "act on" the environment by communicating with the simulation through sockets.

The development of the real-time ObjectAgent architecture is done on the Windows NT computer seen in the background. This computer downlinks OA agents onto the boards and monitors OSE processes in real-time. Ground commands are sent to the spacecraft by sending OSE signals from the NT machine to the real-time boards.

A similar testbed has been set up at the Air Force Research Laboratory (AFRL) at Kirtland Air Force Base and is described in greater detail in Zetocha and Brito [10]. The AFRL testbed was used for the actual demonstration.

Software Agents

For the demonstration, each spacecraft board runs three agents — a sensor agent, a thruster agent, and an orbital trajectory agent. The Cluster Manager spacecraft has an additional reconfiguration agent onboard.

The sensor agent receives spacecraft position, velocity, and remaining fuel from the simulation. The orbital trajectory agent plans a series of thrust commands that will move the spacecraft from the present position to the desired final position. This planning is performed using a simplex linear programming technique. The thruster agent sends these commands to the simulation at the appropriate times.

The reconfiguration agent is responsible for determining where each spacecraft should go upon receipt of a change formation command from the ground. This agent receives the position, velocity, and remaining fuel from the sensor agents on each spacecraft. It uses a version of the algorithm presented by Campbell and Schetter [4] to select the new, desired positions of each spacecraft by optimizing fuel usage across the cluster. These desired positions are then sent to each spacecraft's orbital trajectory agent for the actual maneuver planning. Although these optimization algorithms had been demonstrated in Matlab, they were not available in C++ at the time of the demonstration. Instead, upon receipt of a reconfiguration command from the ground, the reconfiguration agent sent a pre-determined set of new positions to the three spacecraft.

Unfortunately, it is not currently possible to present a useful plot of the results of the simulation. The spacecraft positions can be viewed on the Macintosh screen in real-time and the absolute positions are saved in a text file. However, we are still developing the software tools for analysis and visualization of the outputs.

6. CURRENT STATUS & FUTURE WORK

The Matlab design phase of ObjectAgent is complete and the next version is scheduled for release in December 2000. This version includes more robust and reconfigurable agent communications and relationships, error reporting, and enhanced documentation and examples.

The initial design of the real-time C++ architecture is complete and a demonstration of the reconfiguration of a cluster of three satellites was performed in November 2000.

Following the demonstration, a review of the C++ architecture is being performed and improvements are being made. Work will also begin on the development of a GUI-based tool for the conversion of Matlab agents to C++. This tool and the final C++ architecture should be complete by March 2001. After that time, work will begin in earnest to analyze and debug the various timing and priority issues that will undoubtedly arise. Work will also begin on the watchdog agents for software reliability.

Work is also continuing on the development of the real-time agents that will be used onboard AFRL's TechSat 21 demonstration flight in 2003. TechSat 21 is a mission that will involve three satellites flying in formation and acting as a "virtual" satellite.

ObjectAgent will be used to build two elements of the flight software, the Cluster Manager and the Spacecraft Manager. The Cluster Manager is a flight software package that controls all spacecraft operations that require the coordination of multiple spacecraft. It also provides complete fault detection of all cluster operation related systems. One of the primary functions of the Cluster Manager is to perform relative control of the satellites in the cluster. This will include relative stationkeeping and estimation of the cluster center-of-mass and the relative positions of each satellite.

The Spacecraft Manager is a flight software package that provides an autonomous replacement for the ground operations team. It will control all aspects of spacecraft operation including fault detection and redundancy management. The Spacecraft Manager provides an interface between the Cluster Manager and the rest of the TechSat 21 flight software.

At the conclusion of development, ObjectAgent will provide a robust, easy-to-use software architecture for the control of distributed systems. The flexibility built into the system by the use of agents at all levels enables software to be easily configured and upgraded after deployment. ObjectAgent also provides a common interface to many advanced control and estimation techniques. Its applicability extends beyond clusters of satellites to any real-time, distributed system including robotics, autonomous vehicles, the automobile industry, telecommunications and energy systems, and process control.

7. ACKNOWLEDGMENTS

This work is supported by two United States Air Force SBIR Phase II contracts from the Surveillance and Control Division of the Air Force Research Laboratory's Space Vehicles Directorate. The contract numbers are F29601-99-C-0029 and F29601-00-C-0025 and the program manager is Paul Zetocha.

8. REFERENCES

[1] Bernard, D.E. et al., "Design of the Remote Agent Experiment for Spacecraft Autonomy," *1998 IEEE Aerospace Conference Proceedings*, Snowmass/Aspen, CO, 1998.

[2] Bernard, D.E. et al., "Spacecraft Autonomy Flight Experience: The DS1 Remote Agent Experiment," *1999 AIAA Space Technology Conference & Exposition*, AIAA Paper 99-4512, Albuquerque, NM, September, 1999.

[3] Bradshaw, J.M. (ed.), *Software Agents*, Cambridge, MA: AAAI Press/MIT Press, 1997.

[4] Campbell, M. E. and T. P. Schetter, "Formation Flying Mission for the UW Dawgstar Satellite," *2000 IEEE Aerospace Conference Proceedings*, Big Sky, MT, March, 2000.

[5] Dornheim, M. A., "Deep Space 1 Launch Slips Three Months." *Aviation Week and Space Technology*, April 27, 1998, p. 39.

[6] Knapik, M. and J. Johnson, *Developing Intelligent Agents for Distributed Systems*, New York: McGraw-Hill, 1998.

[7] Schetter, T. P., M. E. Campbell, and D. M. Surka, "Comparison of Multiple Agent-based Organizations for Satellite Constellations," *2000 FLAIRS AI Conference Proceedings*, Orlando, Florida, May 2000.

[8] Schetter, T. P., M. E. Campbell, and D. M. Surka, "Multiple Agent-Based Autonomy for Satellite Constellations," *Second International Symposium on Agent Systems and Applications Proceedings*, Zurich, Switzerland, September 2000.

[9] Weiss, G. (ed.), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, Cambridge, MA: MIT Press, 1999.

[10] Zetocha, P. and M. C. Brito, "Development of a Testbed for Distributed Satellite Command and Control," *2001 IEEE Aerospace Conference Proceedings*, Big Sky, Montana, March 2001.

Derek M. Surka *is the Principal Investigator on TeamAgent and is manager of the DC office of Princeton Satellite Systems. He has worked in the areas of spacecraft autonomy and control since 1994. He received his B.S. from Caltech and S.M. from MIT and is an avid curler.*

Margarita Brito *is an Aerospace Engineer with Princeton Satellite Systems. She is working with others to develop ObjectAgent software to run on the OSE Real Time Operating System. In addition, she is responsible for the integration of ObjectAgent software into the AFRL TechSat 21 Testbed. She holds an MS degree in Aerospace Engineering from MIT.*

Christopher Harvey *is a Software Engineer with Princeton Satellite Systems. He is one of the lead developers of ObjectAgent for the OSE Real Time Operating System. He holds an MS degree in Software Development from Marist College.*