Modeling the LOGOS Multi-Agent System with CSP

Christopher A. Rouff

Science Applications International Corporation
4001 Fairfax Drive, Suite 300
Arlington, Virginia 22203
703-558-8856
rouffc@saic.com

Michael G. Hinchey

NASA Goddard Space Flight Center
Code 500
Greenbelt, Maryland 20771
301-286-9057
mike.hinchey@gsfc.nasa.gov

Abstract

The Lights Out Ground Operations System (LOGOS) system is a prototype multi-agent system for automating satellite ground operations systems. It uses a community of software agents that work cooperatively to perform ground system operations normally done by human operators who are using traditional ground station software tools, such as orbit generators, schedulers and command sequence planners.

During the implementation of LOGOS several errors occurred that involved race conditions. Due to the parallel nature of the asynchronous communications as well as the internal parallelism of the agents themselves, finding the race conditions using normal debugging techniques proved extremely difficult.

Following the development of LOGOS, the development team decided to use formal methods to check for additional inter-agent race conditions and omissions using formal specification techniques. The specification revealed several omissions as well as race conditions. Our experience to date has shown that even at the level of requirements, formalization can help to highlight undesirable behavior and equally importantly can help to find errors of omission.

Introduction

Introducing new and revolutionary technology into a space mission is often met with resistance. From the view points of the Principal Investigator (PI) and the Mission Operations manager, new technology adds risk to an already risky endeavor. A proven success rate for a technology is usually mandated. Agent technology and intelligent software, like other new and emerging technologies, will have an uphill battle for acceptance in operational spacecraft missions.

We are in the process of developing methods and processes for introducing agent-based systems into future satellite missions while both drastically reducing the risk of mission failures and gaining the confidence and support of mission management and PIs. We are doing this by investigating techniques for assuring the correctness of agent software.

The Lights Out Ground Operations System (LOGOS) system is a prototype multi-agent system that was developed at NASA Goddard Space Flight Center (Rouff and Truszkowski 2001) (Truszkowski and Hallock 1999). It is made up of a community of software agents that work cooperatively to perform ground system operations normally done by human operators who are using traditional ground station software tools, such as orbit generators, schedulers and command sequence planners. The agents in LOGOS interface with the existing tools, pass data between each other, perform the needed operations with the above tools and provide services for each other.

In this paper we discuss why and how we applied formal methods to LOGOS to find previously unknown errors and to increase the confidence that PIs and others have in agent and intelligent software systems.

The LOGOS Architecture

For reference, an architecture of LOGOS is shown in Figure 1. LOGOS is made up of 10 agents, some of which interface with legacy software, some which perform services for the other agents in the community, and others which interface with an analyst or operator. All agents can communicate with any other agent in the community, though not all agents need to communicate with each other.

The System Monitoring and Management Agent (SysMMA), keeps track of all of the agents in the community and provides addresses of agents for other agents requesting services. Each agent when started must register with SysMMA to obtain addresses of the other agents and for the other agents to it.

The Fault Isolation and Resolution Expert (FIRE) agent resolves satellite anomalies. FIRE is notified of anomalies during a satellite pass. FIRE contains a knowledge base of potential anomalies and a set of possible fixes for each of the anomalies. If it does not recognize an anomaly or is unable to resolve it, it then has an analyst notified of the problem.

The User Interface Agent (UIFA) is the interface between the graphical user interface that the analyst or operator uses to interact with the LOGOS agent

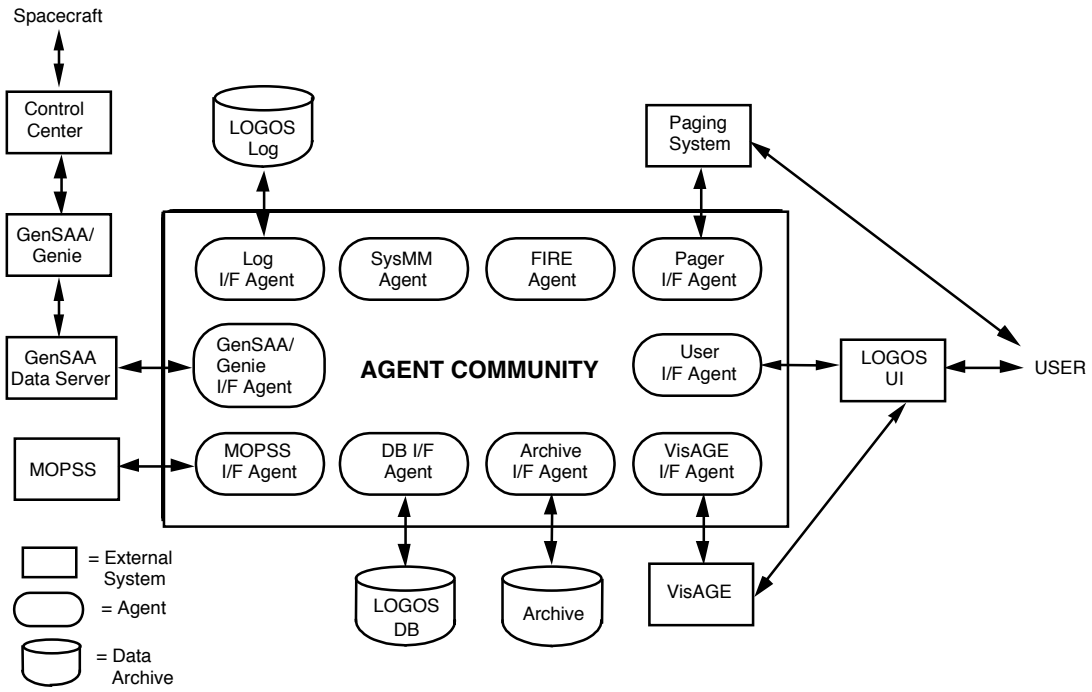

Figure 1: LOGOS agent community and legacy software.

community. UIFA receives notification of anomalies from the FIRE agent, handles login on of users to the system, keeping the user informed with reports, routing commands to be sent to the satellite and other maintenance functions.

The VisAGE Interface Agent (VIFA) interfaces the VisAGE (VisAGE 2000) data visualization system. VisAGE is used to display spacecraft telemetry and agent log information. Real time telemetry information is displayed by VisAGE as it is downloaded during a pass. VIFA requests the data from the GIFA and AIFA agents (see below). An analyst can also use VisAGE to visualize historical information to help monitor spacecraft health or to determine solutions to anomalies or other potential spacecraft problems.

The Pager Interface Agent (PAGER) is the agent community interface to the analyst pager system. If an anomaly occurs or other situation arises that needs an analyst's attention, a request is sent to the PAGER agent which then sends a page to the analyst.

The Database Interface Agent (DBIFA) and the Archive Interface Agent (AIFA) store short term and long term data, respectively, and the Log agent (LOG) stores agent logging data for debugging, illustration and monitoring purposes.

The GenSAA/Genie Interface Agent (GIFA) interfaces with the GenSAA/Genie ground station software, which handles communications with spacecraft. GenSAA/Genie is used to download telemetry data, maintains scheduling information and used to upload commands.

The MOPSS (Mission Operations Planning and Scheduling System) Interface Agent (MIFA) interfaces with the MOPSS ground station planning and scheduling software. MOPSS keeps track of the satellite orbit and when the next pass will be and how long it will last.

An Example Scenario

An example scenario of how the agents would communicate and cooperate would start with MIFA receiving data from the MOPSS scheduling software that the spacecraft LOGOS is controlling will be in contact position in two minutes. MIFA then sends the message to the other agents to let them know of the upcoming event to wake them up, if they are sleeping, and also in case they need to do some preprocessing before the contact. When GIFA receives the message from MIFA it sends a message to the GenSSA Data Server to put it into the proper state to receive transmissions from the control center.

After receiving data, the GenSSA Data Server sends the satellite data to GIFA. GIFA has a set of rules that indicate what data to send to which agents. As well as sending data to other agents, GIFA also sends all engineering data to the archive agent (AIFA) for storage and trend information to the visualization agent (VIFA). Updated schedule information is sent to the scheduling agent (MIFA) and a report is sent to the user interface agent (UIFA) to send on

to an analyst for monitoring purposes. If there are any anomalies, they are sent to the FIRE agent for resolution.

If there is an anomaly, the FIRE agent will try to fix it automatically using a knowledge base containing possible problems and a set of possible resolutions for each problem. To fix an anomaly, FIRE will send a spacecraft command to GIFA to be forwarded on to the spacecraft. After exhausting its knowledge base, if FIRE is not able to fix the anomaly, then FIRE forwards the anomaly to the user interface agent, which then pages an analyst and displays it on their computer for action. The analyst would then formulate a set of commands to send to the spacecraft to resolve the situation. The commands would then be sent to the FIRE agent so it can add the new resolution to its knowledge base for future references and then it sends the commands to the GIFA agent which sends it to the GenSAA/Genie system for forwarding on to the spacecraft.

There are many other interactions going on between the agents and the legacy software, which was not covered above. Examples include the DBIFA requesting user logon information from the database, the AIFA requesting archived telemetry information from the archive database to be sent to the visualization agent, and the pager agent sending paging information to the paging system to alert an analyst of an anomaly needing his or her attention. Additional interactions will be described as needed to illustrate specifications, race conditions or omissions.

Deciding to use Formal Methods

During the implementation of the LOGOS agent community several errors occurred that involved race conditions. Some of these errors included:

- An agent would run correctly on one computer, but when run from another computer would miss a message or work incorrectly.
- Adding additional unrelated agents to the community would cause an agent to crash, miss messages or hang.
- Errors would occur that could not consistently be repeated.
- Buffers would overflow on an inconsistent basis.

Due to the parallel nature of the asynchronous communications as well as the internal parallelism of the agents themselves, finding the race conditions using normal debugging techniques proved extremely difficult, if not impossible. In most cases a through review of the code was performed to manually check for possible problems. Once race conditions were found to be the problem with a few errors, new errors were checked with an eye towards a possible race condition. Code was also identified that could have potential for race conditions and restructured or rewritten to reduce the possibility. This was usually done by removing unnecessary processes or by making unnecessary parallel tasks sequential.

Following the implementation of LOGOS, the development team wanted to check for additional undiscovered errors and explore methods and techniques for checking the correctness of future agent software. The fact that several errors occurred during implementation meant that there might be several more unknown to us. We also knew that the appearance of these errors could be dependent on such non-repeatable conditions such as network congestion.

After looking at several possibilities for finding errors in highly parallel software, the development team decided to use formal methods to see if additional inter-agent race conditions and omissions could be found. Several formal specification languages where investigated and Communicating Sequential Processes (CSP) (Hoare 1985) (Hinchey and Jarvas 1995) was chosen due to it's ease of modeling communicating processes and the availability of local expertise.

The CSP specification of LOGOS brought to light several omissions and race conditions that were not discovered during normal testing (Rouff, Rash and Hinchey 2000). One additional advantage that we found using CSP for LOGOS was its simple structure and how naturally it lends itself to modeling parallel processes, of which LOGOS has many.

Specifying LOGOS

The CSP specification of LOGOS was based on the agent design documents and, when necessary, inspection of the LOGOS Java code. LOGOS is a proof-of-concept system so there were several parts that were specified and designed, but not implemented. The CSP specification reflects what was actually implemented or would be implemented in the near term. A more in depth discussion of this specification can be found in (Rouff, Rash and Hinchey 2000).

In the CSP specification, LOGOS is defined as a process with agents running in parallel as independent processes communicating through an in-house developed software bus, called Workplace, that provides the messaging mechanism between the agents. The definition of LOGOS is given by:

$$\begin{aligned} LOGOS = {} & AIFA \,||\, DBIFA \,||\, FIRE \,||\, GIFA \,||\, LOG \\ & ||\, MIFA \,||\, PAGER \,||\, SysMMA \,||\, UIFA \\ & ||\, VIFA \,||\, Workplace \end{aligned}$$

The above states that each of the agents run in parallel with the other agents and with the WorkPlace communications software. Each agent is able to communication with any other agent, but not all agents need to communicate with each other.

The agents effect their environment and other agents by sending messages over communication channels. Channels are connected to other agents or to the outside environment;

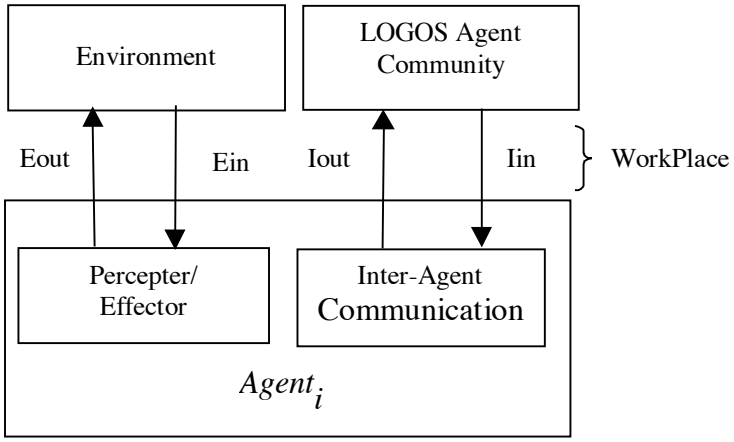

Figure 2: LOGOS agent communication architecture and channels.

the agents do not communicate by direct function call. The channels are connected to either an effector/perceptor component of the agent or the inter-agent communication component (see Figure 2). The effector/perceptor channels are used to connect the agent to its environment.

For channels that communicate with the environment, the channels start with the letter "E". The channels are named Eout for channels sending data to the environment and Ein for channels that receive data from the environment. In the LOGOS context, the environment would consist of the User Interface for the UIFA, the pager system for the PAGER agent, MOPSS for the MIFA, VisAGE for VIFA, the databases for the DBIFA and AIFA, and GenSAA/Genie for GIFA.

For communications with other agents, the agents use the inter-agent communications component and the channels start with the letter "I". Communication between the agents is achieved via message passing based on KQML (Labrou and Finin 1997). Each message has a unique message ID and, when appropriate, an in-reply-to ID that references the original message to which an agent may be replying.

The Iin channel is used to receive messages from other agents and the Iout channel is used to send messages to other agents. Each message is formatted with the name of the agent to which the message is to be sent to, the sending agent, a performative, and parameter/parameter values based on the performative.

The WorkPlace software backplane is used to route the messages to the correct agent on the correct system. For specification purposes we describe the communications taking place over channels. When a message is broadcast on an agent's Iout channel, Workplace will deliver it to the appropriate agent's Iin channel.

From the above, an agent is defined as:

$$Agent_i \triangleq Bus_i \parallel Env_i$$

where each agent has a uniquely defined BUS and ENV process. The BUS process defines the inter-agent communication and the ENV process defines how the agent communicates with entities in the outside environment (such as pagers, databases, and the user interface).

The following sections give two sample specifications from the database interface agent and the user interface agent. An example of a race condition that was found is also provided.

Database Interface Agent Specification

The database interface agent (DBIFA) provides access to a database management system. It stores text on the status and operation of the spacecraft and retrieves information on users. Its definition is:

$$DBIFA \triangleq DBIFA\_BUS \parallel DBIFA\_ENV$$

Which states that the Database agent consists of two processes, the BUS process that communicates with other agents and the ENV process which communicates with the database.

The BUS process for the DBIFA is defined as:

$$\begin{aligned} DBIFA\_BUS = {} & dbifa.Iin?msg \rightarrow \\ & case\ dbifa.Eout!anomaly \rightarrow dbifa.Ein?result \\ & \qquad \rightarrow DBIFA\_BUS \\ & \quad if\ msg = (REQUEST, LOG, anomaly) \\ & dbifa.Eout!telemetry \rightarrow dbifa.Ein?result \\ & \qquad \rightarrow DBIFA\_BUS \end{aligned}$$

$$
\begin{aligned}
&\textit{if } msg = (REQUEST, LOG, telemetry)\\
&dbifa.Eout!sc\_cmd\_sent \rightarrow dbifa.Ein?result\\
&\rightarrow DBIFA\_BUS\\
&\textit{if } msg = (REQUEST, LOG, sc\_cmd\_sent)\\
&dbifa.Eout!sc\_cmd\_exececuted \rightarrow dbifa.Ein?result\\
&\rightarrow DBIFA\_BUS\\
&\textit{if } msg = (REQUEST, LOG, sc\_cmd\_exececuted)\\
&dbifa.Eout!request\_human \rightarrow dbifa.Ein?result\\
&\rightarrow DBIFA\_BUS\\
&\textit{if } msg = (REQUEST, LOG, request\_human)\\
&RETURN\_USER\_SPECIALTY_{\substack{requesting\_agent(msg),\\ anomaly, msg\_id(msg)}}\\
&\textit{if } msg = (REQUEST, RETURN\_DATA, anomaly)\\
&RETURN\_USER\_PASSWORD_{\substack{requesting\_agent(msg),\\ username, msg\_id(msg)}}\\
&\textit{if } msg = (REQUEST, RETURN\_DATA, username)\\
&RETURN\_REPORT_{requesting\_agent(msg), report\_type, msg\_id(msg)}\\
&\textit{if } msg = (REQUEST, RETURN\_DATA, report\_type)\\
&dbifa.Iout!(head(msg), UNRECOGNIZED)\\
&\rightarrow DBIFA\_\text{BUS}\\
&\text{otherwise}
\end{aligned}
$$

The above definition states that the DBIFA_BUS process reads messages from the dbifa.Iin channel and depending on the content of the message executes a particular process. The DBIFA receives several messages from the FIRE agent to log anomalies, telemetry information, commands sent to the spacecraft, commands executed by the spacecraft, and the times when it requests intervention from a user to handle an anomaly. Other messages received by the database agent are requests from the pager agent for pager numbers and from the user interface agent for a user with a particular specialty (to handle an anomaly or other problem), a user's password and a report on the current state of the spacecraft.

If a message is a request to store data in the database, the data is sent to the database on the dbifa.Eout channel, a response is then waited for on the dbifa.Ein channel and then control is given back to the DBIFA_BUS. No confirmation is sent to the requesting agent. Waiting for the database to return comfirmation before continuing is a potential source of deadlock. Also, if there is any error from the database the error is not passed back to the requesting agent. This was due to the prototype nature of LOGOS and should be corrected in future versions.

The first five processes in the above case statement are all requests from agents to store data in the database. The first one is a request to log an anomaly, the second is a request to log a telemetry value, the third is a request to log that the spacecraft was sent a command, the fourth is a request to log that a spacecraft command was executed and the fifth is a request to log that human intervention was requested to solve an anomaly. The last process in the case statement is for the case when a malformed message is based to the agent. In this case the message is returned to the sender with a performative of UNRECOGNIZED.

The remaining three processes are requests for information from other agents and call other processes to handle the requests. The following process definitions for the DBIFA define the responses to the above messages.

$$
\begin{aligned}
&RETURN\_USER\_SPECIALTY_{agent, anomaly, msg\_id} =\\
&\quad dbifa.Eout!(specialist, anomaly)\\
&\rightarrow dbifa.Ein?specialist\_name\\
&\rightarrow dbifa.Iout!(agent, msg\_id, specialist\_name)\\
&\rightarrow DBIFA\_BUS\\
&RETURN\_PAGER\_NUMBER_{agent, specialist, msg\_id} =\\
&\quad dbifa.Eout!(specialist, PAGER)\\
&\rightarrow dbifa.Ein?pager\_number\\
&\rightarrow dbifa.Iout!(agent, msg\_id, pager\_number)\\
&\rightarrow DBIFA\_BUS\\
&RETURN\_USER\_PASSWORD_{agent, username, msg\_id} =\\
&\quad dbifa.Eout!username\\
&\rightarrow dbifa.Ein?(username, password)\\
&\rightarrow dbifa.Iout!(agent, msg\_id, username, password)\\
&\rightarrow DBIFA\_BUS\\
&RETURN\_REPORT_{agent, report-type, msg\_id} =\\
&\quad dbifa.Eout!report\_type\\
&\rightarrow dbifa.Ein?(report\_type, report)\\
&\rightarrow dbifa.Iout!(agent, msg\_id, report\_type, report)\\
&\rightarrow DBIFA\_BUS
\end{aligned}
$$

The processes RETURN_USER_SPECIALTY, RETURN_USER_PASSWORD, and RETURN_REPORT all represent requests to retrieve data from the database. After requesting data from the database they all wait for the data to be returned before continuing. This is a situation that can lead to a deadlock state since there is no timeout mechanism. Once the response from the database is received over the dbifa.Ein channel, the data is sent back to the requesting agent over the dbifa.Iout channel along with the original message id.

Since the database agent does not receive any unsolicited messages from the database system (all messages between the database and DBIFA are synchronized on), the DBIFA_ENV process does not need to be defined, and is therefore left undefined.

User Interface Specification

The user interface agent (UIFA) provides a connection between the user interface and the LOGOS agent community. The UIFA receives commands from the user

interface and passes those commands to the appropriate agent and also receives data from other agents and sends that information to the user interface.

Like the other agents, the UIFA specification consists of a BUS and ENV process:

$$UIFA \triangleq UIFA\_BUS_{[\,],[\,],\{\,\}} || UIFA\_ENV$$

The first bag parameter represents the requests for user passwords that have been made to the database agent and the second bag holds the names of the users that have logged on. The reason why they are bags instead of sets is because a single user can be logged on multiple times simultaneously. The third parameter is the set of anomalies that are waiting to be sent to the user interface when there are no users logged on. It is a set because duplicate anomalies are only reported once. The three bags/sets are initialized to empty because, when LOGOS starts up, no users should be logged in and no anomalies have been recorded.

UIFA BUS Process

The specification of the UIFA_BUS process is:

$$UIFA\_BUS_{db\_waiting,logged\_in,anomalies} \triangleq$$
$$UIFA\_INTERNAL_{db\_waiting,logged\_in,anomalies}$$
$$| UIFA\_AGENT_{db\_waiting,logged\_in,anomalies}$$

The above states that the UIFA_BUS process can either be the UIFA_AGENT process or the UIFA_INTERNAL process. The UIFA_AGENT process defines the response UIFA makes to messages received from other agents over the uifa.Iin channel. These messages are requests from other agents to send the user interface data or are responses to requests that the UIFA made earlier to another agent. The UIFA_INTERNAL process is used to transfer password related data from the UIFA_ENV process to the UIFA_BUS.

The definition for the UIFA_INTERNAL process is rather simple and is solely for transferring password information received from the user in the UIFA_ENV process to the database agent for validation. It also needs to store the password information in the db_waiting parameter for future retrieval by the UIFA_BUS process when the database agent returns the password validation information (as will be seen below). The UIFA_INTERNAL process is defined as:

$$UIFA\_INTERNAL_{db\_waiting,logged\_in,anomalies} =$$
$$uifa.pass?(user\_name, password)$$
$$\rightarrow uifa.Iout!(DBIFA, RETURN\_DATA,$$
$$user\_name, password)$$
$$\rightarrow UIFA\_BUS_{db\_waiting \cup (user\_name,password),\ logged\_in,anomalies}$$

which states that when a user name/password tuple is sent from the UIFA_ENV process over the uia.pass channel, this process sends the tuple to the database agent for validation and then calls the UIFA_BUS process again with the (user_name, password) tuple added to the db_waiting bag. Later, when the database agent returns with its stored password for this user, it will be compared with the user name and password received from the user interface and stored in db_waiting.

The following gives a partial specification of the UIFA_AGENT process and how it handles messages received from other agents.

$$UIFA\_AGENT_{db\_waiting,logged\_in,anomalies} = uifa.Iin?msg \rightarrow$$
$$\text{case } RESOLVE\_ANOMALY_{db\_waiting,logged\_in,\ anomailes,new\_anomaly}$$
$$\text{if } msg = (REQUEST, RESOLVE, new\_anomaly)$$
$$RECEIVED\_SPECIALIST_{db\_waiting,\ logged\_in,\ anomalies,new\_anomaly,\ specialist}$$
$$\text{if } msg = (RETURN\_DATA, specialist)$$
$$RECEIVED\_PASSWORD_{db\_waiting,logged\_in,anomalies,\ user\_name,\ db\_password}$$
$$\text{if } msg = (RETURN\_DATA, user\_name, db\_password)$$

$$uifa.Eout!(REFERRED, referred\_anomalies)$$
$$\rightarrow UIFA\_BUS_{db\_waiting,logged\_in,anomalies}$$
$$\text{if } msg = (RETURN\_DATA, referred\_anomalies)$$
$$\wedge\ logged\_in \neq []$$

$$uifa.Eout!(user\_name, INVALID)$$
$$\rightarrow UIFA\_BUS_{db\_waiting/(user\_name,y)\ \bullet\ (\exists!(user\_name,y) \in db\_waiting),\ logged\_in,anomalies}$$
$$\text{if } msg = (USER\_EXCEPTION, user\_name, exception)$$
$$\wedge\ logged\_in \neq []$$

$$uifa.Iout!(head(msg), UNRECOGNIZED)$$
$$\rightarrow UIFA\_BUS_{db\_waiting,logged\_in,anomalies}$$
$$otherwise$$

The first element of the above case statement is a request for the user to solve an anomaly, which is processed by UIFA_RESOLVE_ANOMALY (described below). The REFERRED process sends the user interface a list of anomalies the FIRE agent has sent the user interface that have not yet been fixed. The USER_EXCEPTION process indicates that the user name does not exist and sends an invalid log-in message to the user interface. The processes named RESOLVE_ANOMALY, RECEIVED_PASSWORD and RECEIVED_SPECIALIST need to test state information before passing data on to the user interface.

RESOLVE_ANOMALY checks whether a user is logged on before passing the anomaly to the user interface. If a user is logged on, the anomaly is sent on to the user interface. If a user is not logged on, then a user that has the required subsystem specialty is paged. The paging is done by:

1. Requesting the specialist type needed for this type of anomaly from the database.
2. Once the data is received from the database, sending the pager the specialist type to page.
3. And then waiting for the user to log in.

When a user logs in and the password is received from the database and validated, the process checks whether any anomalies have occurred since the last log in. If anomalies exist, they are sent to the user interface after the user interface has been informed that the user's name and password have been validated.

The following is the specification of what the UIFA does when it receives a request to send the user interface an anomaly.

$$RESOLVE\_ANOMALY_{db\_waiting,\text{logged\_in},\ anomalies,new\_anomaly} =$$
$$uifa.Eout!new\_anomaly$$
$$\rightarrow UIFA\_BUS_{db\_waiting,\text{logged\_in},anomalies}$$
$$\quad if\ \ \text{logged\_in} \neq [\ ]$$
$$else$$
$$uifa.Iout(DBIFA,REQUEST,SPECIALIST,$$
$$\quad new\_anomaly)$$
$$\rightarrow UIFA\_BUS_{db\_waiting,\text{logged\_in},anomalies \cup new\_anomaly}$$

UIFA_RESOLVE_ANOMALY states that when the UIFA receives an anomaly it first checks whether the user is logged on by checking the list of users in the bag "*logged_in*". If the user is logged on, then the anomaly is sent to the user interface over the environment channel uifa.Eout. If the user is not logged on, then UIFA sends a message to the database asking it for a specialist that can handle the given anomaly (the returned information will later be sent to the pager).

The following is the specification as to what happens when the specialist type is received from the database agent.

$$RECEIVED\_SPECIALIST_{db\_waiting,logged\_in,anomalies,\ new\_anomaly,\ specialist} =$$
$$\rightarrow uifa.Iout!(PAGER,specialist,new\_anomaly)$$
$$\quad if\ \ logged\_in = []$$
$$\rightarrow UIFA\_BUS_{db\_waiting,logged\_in,anomalies}$$

RECEIVED_SPECIALIST states that when the name of the specialist is received from the database, the pager is sent the type of specialist to page. If someone has logged on since the request to the database was made, then the specialist is not sent to the pager and nothing further is done (the anomaly was sent to the logged on user).

The following specifies what happens when a password is received from the user interface:

$$RECEIVED\_PASSWORD_{db\_waiting,\ \text{logged\_in},\ anomalies,user,db\_password} =$$
$$uifa.Eout!(user,\ VALID)$$
$$\rightarrow CHECK\_ANOMALIES_{db\_waiting/(user,db\_password),\ \text{logged\_in} \cup (user),anomalies}$$
$$\quad if\ (user,\ db\_password) \in db\_waiting\ \ else$$
$$\rightarrow uifa.Eout!(user,INVALID)$$
$$\rightarrow UIFA\_BUS_{db\_waiting/(user,y)\ \bullet\ (\exists!(user,y) \in db\_waiting),\ \text{logged\_in},anomalies}$$

The above password validation process, RECEIVED_PASSWORD, is executed when the database agent returns the password requested by UIFA_ENV as described below. The process executed after RECEIVED_PASSWORD is dependent on whether the password is valid or not. The password is validated by checking whether the tuple (user name, password) received from the database is in the bag db_waiting. If it is, a VALID message is sent to the user interface and the CHECK_ANOMALIES process is called with the (user name, password) tuple removed from the db_waiting bag and added to the logged_in bag. If the (user name,password) tuple is not in db_waiting, an INVALID message is sent to the user interface and the UIFA_BUS process is called with the (user name, password) tuple in db_waiting removed.

The following are the processes executed when the user name and password are validated:

$$CHECK\_ANOMALIES_{db\_waiting,logged\_in,anomalies} =$$
$$uifa.Iout(PAGER,STOP\_PAGING)$$
$$\rightarrow SEND\_ANOMALIES_{db\_waiting,\ logged\_in,anomalies}$$
$$\quad if\ \ anomalies \neq \{\},\ \ else$$
$$\rightarrow UIFA\_BUS_{db\_waiting,logged\_in,anomalies}$$

$$SEND\_ANOMALIES_{db\_waiting,logged\_in,a \cup A} =$$
$$uifa.Eout!a\ \ \rightarrow SEND\_ANOMALIES_{db\_waiting,logged\_in,A}$$

$$SEND\_ANOMALIES_{db\_waiting,logged\_in,a \cup \{\}} =$$
$$uifa.Eout!a\ \ \rightarrow\ UIFA\_BUS_{db\_waiting,logged\_in,\{\}}$$

In this process, the anomalies set is checked to see whether it is non-empty, and if it is, then the process SEND_ANOMALIES is called. If the anomalies set is

empty, then the UIFA_BUS process is executed. The SEND_ANOMALIES process sends each of the anomalies in the anomalies set to the user interface and deletes each from the set. When the anomalies set is empty, it calls the UIFA_BUS process again.

UIFA ENV Process

The ENV process for the UIFA reads messages from the user interface and processes them or passes them on to other agents as needed. It is defined as:

```
UIFA_ENV = uifa.Ein?msg →
  case uifa.pass!(user_name,password)   → UIFA_ENV
          if  msg = (LOGIN,user_name,password)
    uifa.Iout!(MIFA,REQUEST,SCHEDULE)  → UIFA_ENV
          if  msg = (GET_SCHEDULE)

    uifa.Iout!(DBIFA,REQUEST,REPORT)  → UIFA_ENV
          if  msg = (GET_CURRENT_REPORT)
    uifa.Iout!(FIRE,SC_COMMAND,command)  → UIFA_ENV
          if  msg=(SC_COMMAND,command)
```

The above definition for the UIFA_ENV process states that it reads a message off of the environment channel *uifa.Ein* (from the user interface) and stores it in the *msg* variable. The process that is executed next is dependent on the content of the message from the user interface. If it is a user name and password, then it is sent over the *uifa.pass* channel to the UIFA_BUS process, which then sends the user name and password to the database agent (see above).

The other three possibilities in the case statement are requests from the user for the current schedule, a report or a command to send to the spacecraft. In each of these three cases the message is sent to the indicated agent over the *uifa.Iout* channel with the request.

Race Conditions

While specifying the user interface agent, we discovered that a race condition exists between the RESOLVE_ANOMALY process and the RECEIVED_PASSWORD process. The race condition can occur if an anomaly occurs at about the same time a user logs in, but before the user is completely logged in. The race condition can occur in several ways. The following scenario illustrates one of them:

1. A user logs in.
2. The user interface passes the user name and password to the user interface agent (UIFA).
3. UIFA sends the database agent (DBIFA) the user name, requesting a return of the password.
4. While UIFA is waiting for the DBIFA to send the password back, the FIRE agent sends UIFA an anomaly to send to the user.
5. A process in UIFA checks to see whether the user is logged on, which is not the case, but, before it can set the variable to indicate an anomaly is waiting, the process blocks.
6. At this point, the password is received from the database, and this UIFA process determines that the password is valid and checks to see whether there are any anomalies waiting, which is not the case (because the process in #5 is blocked and has not set the variable yet), and then sends a valid login message to the user interface.
7. At this point the process in #5 above becomes unblocked and continues; it then finishes setting the variable indicating that an anomaly is waiting, has the user paged, and then blocks waiting for a new user to log in.

Although the above situation is by no means fatal since in the end a user is paged, nevertheless, a user is paged even though one who could handle the anomaly immediately is already logged on. In addition, if the anomaly needed an immediate response, time would be wasted while the second user responded to the page.

From the above, it is evident that this is an invariant condition for UIFA. The invariant is that the anomaly-waiting variable and the user-logged-in variable should never be set to true at the same time. If this invariant is violated, then the condition exists that a user is logged in, the anomaly process has unnecessarily paged a user and is waiting for that user to login instead of sending the anomaly to the current user.

This condition can be fixed fairly easily. In each agent there is a process that executes regularly to do any needed housekeeping functions. Some code could be added that checks whether the anomaly-waiting variable is set at the same time the user-logged-in variable is set and then calls the appropriate routine to send the anomaly to the user and unset the anomaly-waiting variable.

Another potential race condition occurs when a user logs in and enters an incorrect password. When the database returns the correct password to the UIFA, the UIFA has in its bag the name of the user and the entered password. If this password is incorrect, it normally can be easily deleted, because the element in the bag can be indexed by the user name. A problem can arise if the password is incorrect and if, before the database has a chance to reply, the same user logs in again with a different password. In this case, when the database returns the correct password from the first occurrence, it will not know which (username, password) tuple to delete.

The solution to this problem is also straight forward. The message id of the request to the database should be used as the key and not the username, since the message id is unique. The message id will also be returned by the database in the “in-reply-to” field of its message. So instead of storing the couple (username, password) in db_waiting, the triple (message id, username, password) should be used.

Conclusion

Our experience to has shown that even at the level of requirements, formalization in CSP can help to highlight undesirable behavior and errors of omission. We are currently working on using formal and semi-formal techniques for specifying our next generation multi-agent system to check for possible error conditions (including deadlock) between components of the agents and between the agents themselves.

We have found that the results that formal specifications can provide are extremely helpful, but developing them is very time intensive, and the results may not be available until after the project has been completed (as in this case). We have found that tools are essential to speeding up this process. We are currently experimenting with model checkers such as Spin (Holzmann 1991) and in the future the Java PathFinder (Havelund 1999) for checking existing Java code for errors. We are also actively researching the development of other tools that can speed the process of developing formal specifications and checking these specifications for errors.

As programming languages such as Java make it easier to develop concurrent systems, the need to find race conditions and other concurrency related errors will become even more important. The only cost-effective way that has been described for doing this with any assurance requires the use of formal methods. We believe that the development of tools will ease the process of learning and using formal methods.

Acknowledgments

Our thanks to the LOGOS development team for their help in understanding the inner workings of LOGOS. The development team includes: Walt Truszkowski, James Rash, Tom Grubb, Troy Ames, Carl Hostetter, Jeff Hosler, Matt Brandt, Dave Kocur, Kevin Stewart, Jay Karlin, Victoria Yoon, Chariya Peterson, and Dave Zock, who are or have been members of the Goddard Agent Group.

References

Havelund, K. 1999. Model Checking Java Programs using Java PathFinder. International Journal on Software Tools for Technology Transfer.

Hinchey, M.G. and Jarvis, S.A. 1995. *Concurrent Systems: Formal Development in CSP*. McGraw-Hill International Series in Software Engineering. London and New York.

Holzmann, H. J. 1991. *Design and Validation of Computer Protocols*. Prentice Hall Software Series. Englewood Cliffs, NJ.

Hoare, C.A.R. 1985. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, Hemel Hempstead.

Labrou, Y. and Finin, T. 1997. A Proposal for a new KQML Specification. Technical Report TR CS-97-03. Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250.

Rouff, C., Rash, J., and Hinchey, M. 2000. Experience Using Formal Methods for Specifying a Multi-Agent System. In Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2000).

Rouff, C., and Truszkowski, W. 2001. A Process for Introducing Agent Technology into Space Missions. In Proceedings of the IEEE Aerospace Conference.

Truszkowski, W., Hallock, H. 1999. Agent Technology from a NASA Perspective. CIA-99, Third International Workshop on Cooperative Information Agents. Springer-Verlag, Uppsala, Sweden.

VisAGE 3.0 User's Manual. 2000. NASA Goddard Space Flight Center. Code 588.0, http://tidalwave.gsfc.nasa.gov/avatar/.