Experience Using Formal Methods for Specifying a Multi-Agent System

Christopher Rouff, James Rash
NASA Goddard Space Flight Center
Code 588.0
Greenbelt, MD 20771
chris.rouff@gsfc.nasa.gov,
james.rash@gsfc.nasa.gov

Michael Hinchey
Computer Science Department
University of Nebraska, Omaha
Omaha, NE 68182
mike@ida.his.se

Abstract

The process and results of using formal methods to specify the Lights Out Ground Operations System (LOGOS) is presented in this paper. LOGOS is a prototype multi-agent system developed to show the feasibility of providing autonomy to satellite ground operations functions at NASA Goddard Space Flight Center (GSFC).

After the initial implementation of LOGOS the development team decided to use formal methods to check for race conditions, deadlocks and omissions. The specification exercise revealed several omissions as well as race conditions. After completing the specification, the team concluded that certain tools would have made the specification process easier.

This paper gives a sample specification of two of the agents in the LOGOS system and examples of omissions and race conditions found. It concludes with describing an architecture of tools that would better support the future specification of agents and other concurrent systems.

1. Introduction

Space missions until recently have been operated manually from ground control centers. The high costs of satellite operations has prompted NASA and other funding sources to seriously look into automating as many functions as possible. A number of more-or-less automated ground operation systems exist today, but work continues with the goal of reducing system costs to even lower levels.

Cost reduction can be achieved in a number of areas. Greater autonomy of satellite ground control functions is one area that can greatly reduce the overall cost of missions. The Lights Out Ground Operations System is (LOGOS) [5] is a proof of concept system that uses a community of autonomous software agents that work cooperatively to perform the functions previously done by human operators who were using traditional software tools, such as orbit generators and command sequence planners.

Figure 1 shows the architecture of LOGOS. It consists of ten agents that interact with each other and external ground control software. The following is a brief description of each of the agents:

FIRE: the Fault Isolation and Resolution Expert Agent resolves satellite anomalies,
DBIFA: the Database Interface Agent interfaces with a database management system for storage of short term data,
UIFA: the User Interface Agent transfers data to and commands from the user interface,
PAGER: an agent that pages users when the satellite needs attention and LOGOS is unable to determine the needed response,
AIFA: the Archive Interface Agent stores and retrieves telemetry and other long term data,
MIFA: the Mission Operations Planning and Scheduling System (MOPSS) Interface agent produces satellite scheduling information for the other agents,
SysMMA: the System Monitoring and Management Agent is where each of the agents in the community register with and queries for names of agents with particular abilities,
VIFA: the Visage Interface agent passes data to the Visage visualization system to display telemetry data,
LOG: is an agent that logs status messages from other agents and sends them to the user interface for display,
GIFA: the Generic Spacecraft Analyst Assistant (GenSAA) Interface Agent interfaces with GenSAA for transferring data to and from spacecraft.

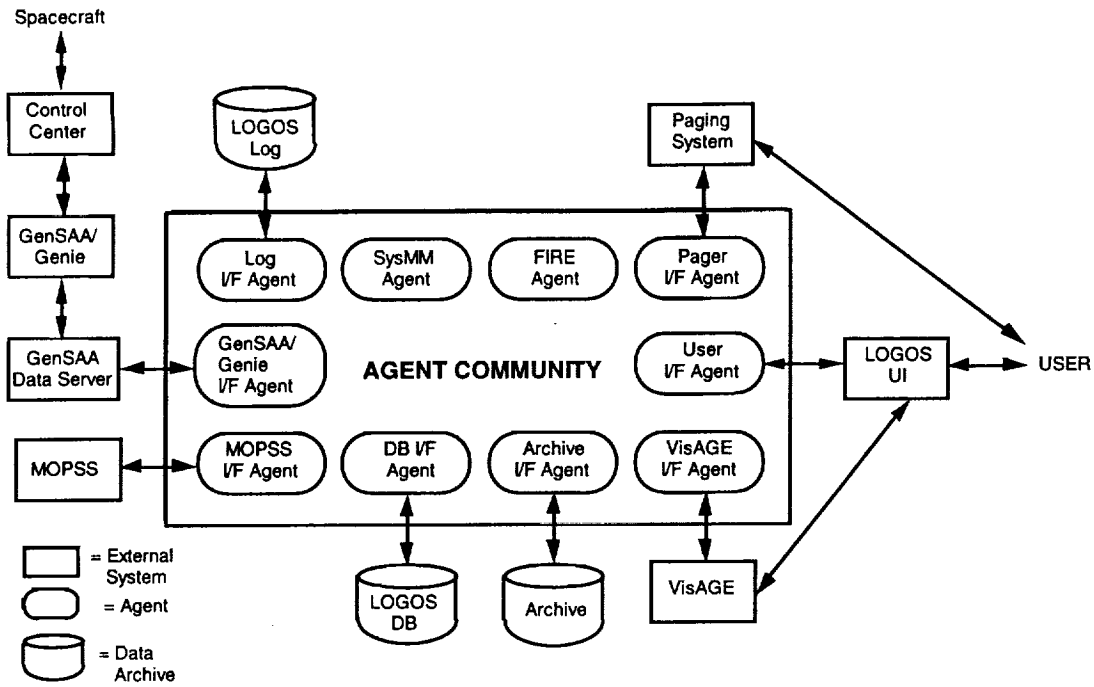

Figure 1: LOGOS Architecture

2. LOGOS Specification

The specification of the LOGOS agents was done using Communicating Sequential Processes (CSP) [2] [4]. One advantage that we found using CSP for LOGOS was it's simple structure and it's naturalness in modeling parallel processes, which LOGOS has many.

The CSP specification of LOGOS is based on the agent design documents and when necessary inspection of the LOGOS code. LOGOS is a proof of concept system so there were several parts that were specified, designed, but not implemented. The specification reflects what was actually implemented or would be implemented in the near term. If a feature was not implemented the intended implementation was specified based on the design document as it was to be completed in the near term or in the second generation of LOGOS.

In the CSP specification LOGOS is defined as a process with agents running in parallel as independent processes using an in-house developed software bus, called Workplace, that provides the communications mechanism between the agents. LOGOS's definition is:

$$LOGOS = AIFA \parallel DBIFA \parallel FIRE \parallel GIFA \parallel LOG \parallel MIFA \parallel PAGER \parallel SysMMA \parallel UIFA \parallel VIFA \parallel WorkPlace$$

The agents produce effects by communicating via their output channels to other agents or to the outside environment, not by direct action. For example, to cause a spacecraft to execute a command, an agent would send an appropriate command via the spacecraft interface agent, which in turn would send a command data packet to the appropriate spacecraft commanding software (in this case GenSAA). Communication between the agents is done through messages. 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.

Workplace is used to transport the messages between

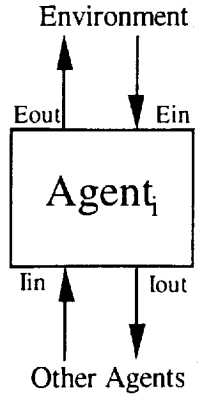

Figure 2: Channels agents communicate over.

agents. Each agent has "in" and "out" channels (see Figure 2) connected to Workplace for sending and receiving messages. Some of the agents are interface agents that communicate with an outside environment (i.e., the user interface, scheduler, pager, etc.). These interface communications are implemented by each agent individually and can also be thought of as having ``in" and ``out" channels to the agent's environment. Agents that do not communicate with an outside environment are considered to have null communications over their environment channels.

From the above, an agent is defined as:

$$Agent_i \triangleq Bus_i \,||\, Env_i$$

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 the outside environment (such as pagers, databases, and the user interface).

Each agent also communicates over four channels: Ein, Eout, Iin, and Iout. The Ein and Eout channels provide the input and output to the agent's environment in the Env processes and the channels Iin and Iout provide the input and output communications to the other agents via WorkPlace in the Bus processes. When a message is broadcast on an agent's Iout channel, Workplace will deliver it to the appropriate agent's Iin channel.

Messages sent between agents contain a predefined header and data consisting of:

- the target agents name,
- the sending agents name,
- a message ID,
- an in-reply-to message ID if appropriate,
- a message type,
- a performative, and
- combinations of parameter names and parameter values.

For the purpose of brevity, non-essential parameters are not included in the specification. In *case* and *if* statements, a message is matched when the stated parameters in the message are matched. In addition, capitalized words indicate constants that are being matched, and lower or mixed case words represent variables that contain the indicated parameter value. Many of the error conditions are not shown to increase the readability of the specification.

3. Pager Agent Specification

The following is the specification of the pager agent. The Pager agent sends pages to engineers and controllers when there is a spacecraft anomaly and the FIRE agent can not figure out a solution and there is no one logged on to notify that an anomaly has occured. The Pager agent receives requests from the user interface agent (UIFA), gets paging information from the database agent, and when instructed by the UIFA, stops paging. The pager agent is defined as:

$$PAGER \triangleq PAGER\_BUS_{\{\},\{\}} \,||\, PAGER\_ENV$$

As discussed above, the Pager agent is defined as a process, PAGER, which is also defined as two processes running in parallel: a BUS process and an ENV process. The first empty set in the parameter list for PAGER_BUS represents the list of requests made to the database agent for paging information on a particular specialist. The second empty set represents the list of specialists that are currently being paged. The paged parameter is a set and not a bag because even though someone can be paged multiple times, a stop page command only has to be sent one time, not the number of times the specialist was paged (it is assumed the specialist only has one pager). Since the pager agent initially starts out with no requests for pages, these sets are initialized to be empty.

The above bus and environment processes for an agent are defined in terms of their input events. The BUS process is defined as:

$$PAGER\_BUS_{db\_waiting,paged} = pager.Iin?msg \rightarrow$$

case $GET\_USER\_INFO_{db\_waiting,paged,pagee,text}$

if $msg = (START\_PAGING, specialist, text)$

$BEGIN\_PAGING_{db\_waiting,paged,\ in\_reply\_to\_id(msg),pager\_num}$

if $msg = (RETURN\_DATA, pager\_num)$

$STOP\_CONTACT_{db\_waiting,paged,pagee}$

if $msg = (STOP\_PAGING, pagee)$

$pager.Iout!(head(msg), UNRECOGNIZED)$

$\rightarrow PAGER\_BUS_{db\_waiting,paged}$

otherwise

The above specification states that the process PAGER_BUS receives a message on it's "Iin" channel (the pager's in channel from WorkPlace) and stores it in a variable called "msg". Depending on the contents of the message, one of four different processes are executed.

If the message has a START_PAGING performative, then the GET_USER_INFO process is called with parameters of the type of specialist to page (pagee) and

4.1 UIFA BUS Process

The specification of the UIFA_BUS process is:

$$UIFA\_BUS_{db\_waiting,\log ged\_in,anomalies} \triangleq$$
$$UIFA\_INTERNAL_{db\_waiting,\log ged\_in,anomalies}$$
$$\mid UIFA\_AGENT_{db\_waiting,\log ged\_in,anomalies}$$

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:

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

and 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 it's 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 specifies the UIFA_AGENT process and how it handles messages received from other agents.

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

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

$$uifa.Eout!BEGIN\_ACTIVITY$$
$$\rightarrow UIFA\_BUS_{db\_waiting,\log ged\_in,anomalies}$$
$$\text{if } msg = (BEGIN\_ACTIVITY) \wedge logged\_in \neq []$$

$$uifa.Eout!END\_ACTIVITY$$
$$\rightarrow \text{UIFA\_BUS}_{db\_waiting,\log ged\_in,anomalies}$$
$$\text{if } msg = (END\_ACTIVITY) \wedge logged\_in \neq []$$

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

$$uifa.Eout!(REPORT, report)$$
$$\rightarrow \text{UIFA\_BUS}_{db\_waiting,\log ged\_in,anomalies}$$
$$\text{if } msg = (REPORT, report) \wedge logged\_in \neq []$$

$$uifa.Eout!(REPORT\_EXCEPTION, exception)$$
$$\rightarrow UIFA\_BUS_{db\_waiting,\log ged\_in,anomalies}$$
$$\text{if } msg = (REPORT, exception) \wedge logged\_in \neq []$$

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

$$uifa.Eout!(SCHEDULE, schedule)$$
$$\rightarrow UIFA\_BUS_{db\_waiting,\log ged\_in,anomalies}$$
$$\text{if msg} = (\text{SCHEDULE, schedule}) \wedge logged\_in \neq []$$

$$uifa.Eout!(SCHEDULE\_UPDATE, schedule)$$
$$\rightarrow UIFA\_BUS_{db\_waiting,\log ged\_in,anomalies}$$
$$\text{if msg} = (\text{UPDATE\_DATA, schedule}) \wedge logged\_in \neq []$$

$uifa.Iout!(head(msg), UNRECOGNIZED)$

$\rightarrow UIFA\_BUS_{db\_waiting, logged\_in, anomalies}$

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 below. The rest of the messages received by the UIFA are replies to requested data or information on events that occurred and are simply to be passed on to the user interface, if a user is logged on. The LOGOS requirements document currently does not allow multiple users, therefore keeping track of which user requested which information and to which user anomalies should be sent to for resolution is not specified.

The case statements with the following performatives in their case statement are information messages stating a particular event occurred: BEGIN_ACTIVITY, END_ACTIVITY, and SCHEDULE_UPDATE. The BEGIN and END activities represent acquisition of a signal from the satellite and the loss of that signal. The performative SCHEDULE_UPDATE is a notice that the satellite's schedule has changed and a new version is included in the message. The remainder of the messages are all replies to requests that UIFA made to other agents and simply pass the data on to the user interface over the uifa.Eout channel (if a user is logged in, otherwise it is ignored). The following is a short description of each of the above processes, listed by their performative:

- UNRESOLVED process sends a list of anomalies that the FIRE agent is still working on to the user interface over the uifa.Eout channel.
- REFERRED process sends the user interface a list of anomalies the FIRE agent has sent the user interface that have not yet been fixed.
- BEGIN_ACTIVITY and UIFA_END_ACTIVITY process send a begin/end activity message on to the user interface from the SysMAA agent.
- REPORT sends a report from the database to the user interface.
- REPORT_EXCEPTION process sends a report exception message to the user interface indicating there are no reports available.
- USER_EXCEPTION process indicates that the user name does not exist and sends an invalid log in message to the user interface.
- SCHEDULE sends a schedule from MOPSS agent to the user interface.
- SCHEDULE_UPDATE sends a message to the user interface that the schedule has been updated.

Processes UIFA_RESOLVE_ANOMALY, UIFA_RECEIVED_PASSWORD and UIFA_RECEIVED_SPECIALIST have to test state information before passing data on to the user interface. UIFA_RESOLVE_ANOMALY checks if 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 (see also the PAGER specification above), and
3. then waiting for the user to log in.

When a user logs in and the password is received from the database and validated, a check is also made if any anomalies have occurred since the last log in. If there are anomalies, the anomaly is sent to the user interface after the user interface has been informed that the user's name and password has been validated.

The following is the specification for when UIFA receives a request to send the user interface an anomaly.

$RESOLVE\_ANOMALY_{db\_waiting, logged\_in, anomalies, new\_anomaly} =$

$uifa.Eout!new\_anomaly$

$\rightarrow UIFA\_BUS_{db\_waiting, logged\_in, anomalies}$

if $logged\_in \neq [\,]$

else

uifa.Iout(DBIFA, REQUEST, SPECIALIST, new_anomaly)

$\rightarrow UIFA\_BUS_{db\_waiting, logged\_in, anomalies \cup new\_anomaly}$

UIFA_RESOLVE_ANOMALY states that when the UIFA receives an anomaly notice, 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 over the environment channel uifa.Eout to the user interface. 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 (later the returned information will be sent to the pager).

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

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

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 is the specification for when a password is received from the user interface:

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

The above password validation process, UIFA_RECEIVED_PASSWORD, is executed when the database agent returns the password requested below in UIFA_ENV. The process executed after UIFA_RECEIVED_PASSWORD is dependent on whether the password is valid or not. The password is verified by checking if 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:

$$CHECK\_ANOMALIES_{db\_waiting,\log ged\_in,anomalies} =$$
$$uifa.Iout(PAGER, STOP\_PAGING)$$
$$\rightarrow \text{SEND\_ANOMALIES}_{db\_waiting,\ logged\_in,\ anomalies}$$
$$if\ \text{anomalies} \neq \{\},\ \text{else}$$
$$\rightarrow \text{UIFA\_BUS}_{db\_waiting,\ logged\_in,\ anomalies}$$

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

$$SEND\_ANOMALIES_{db\_waiting,\log ged\_in,a\cup\{\}} =$$
$$uifa.Eout!a \rightarrow \text{UIFA\_BUS}_{db\_waiting,logged\_in,\{\}}$$

In this process the anomalies set is check to see if 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.

4.2 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:

$$UIFA\_ENV = uifa.Ein?msg \rightarrow$$
$$case\ uifa.pass!(user\_name, password) \rightarrow UIFA\_ENV$$
$$if\ msg = (\text{LOGIN, user\_name, password})$$
$$uifa.Iout!(MIFA, REQUEST, SCHEDULE) \rightarrow UIFA\_ENV$$
$$if\ msg = (\text{GET\_SCHEDULE})$$
$$uifa.Iout!(DBIFA, REQUEST, REPORT) \rightarrow UIFA\_ENV$$
$$if\ msg = (GET\_CURRENT\_REPORT)$$
$$uifa.Iout!(FIRE, \text{SC\_}COMMAND, command) \rightarrow UIFA\_ENV$$
$$if\ msg = (SC\_COMMAND, command)$$

The above definition for the UIFA_ENV process states that it reads messages 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, 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.

5. Omissions and Race Conditions

The following describes some of the omissions and deadlock conditions we found in LOGOS by specifying it using CSP.

5.1 Omissions

Due to the prototype nature of LOGOS there were many omissions found. The omissions were usually one of two types:

- What an agent is to do when another agent never responds to a request
- How long an agent should wait for another agent to respond to a request

The following is some samples of omissions from the pager agent:

- should the pager agent automatically resubmit a page if there has been no response by a specified amount of time or should this command come from the user interface agent
- should the pager agent change who they are paging after an elapsed period of time, and what should that time interval be. Again, or should that information come from the UIFA.
- what happens when the pager agent receives a request to page someone that it has already been paged (i.e., the pager has not received a request yet to stop paging the party that is being requested to page). In this situation the pager agent can either re-page the party or ignore the request. In addition, if a party can be paged multiple times, does the software have to keep track of the number of times the party was paged or other relevant information.
- should the pager agent cache specialist pager numbers and information for a specific amount of time, or should they always be requested from the database (even if there is an active, unanswered page for a specialist).
- There is nothing specified as to what to do if the requested pagee does not exist.

5.2 Race Conditions

While specifying the user interface agent it was discovered that a race condition exists between the UIFA_RESOLVE_ANOMALY process and the UIFA_RECEIVED_PASSWORD process. The way the race condition can happen is if an anomaly occurs 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 if the user is logged on, which it is not, 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, this UIFA process determines that the password is valid, checks to see if there are any anomalies waiting, which there are not (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, none the less, a user is paged even though one is logged on and could handle the anomaly immediately. 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 on variable is set and call the appropriate routine to send the anomaly to the user and unset the anomaly-waiting variable.

Another potential race condition is 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. The problem that can come up is 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 fairly simple. 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 it's message. So instead of a two tuple of (username, password) being stored in db_waiting, the three-tuple (message id, username, passowrd) should be used.

6. Conclusion

Our experience to date has shown that even at the level of requirements, formalization in CSP can help to highlight undesirable behavior and equally importantly can help to point out 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 the case presented here). We have found that tools are essential to speeding up this process. We are currently experimenting with model checkers, such as Spin [3], and the Java PathFinder [1] (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 way to do this with any assurance will be through formal methods. We believe that the use of tools will ease the process of learning and using formal methods.

7. Acknowledgements

Our thanks to the LOGOS development team for their help in us understanding the inner workings of LOGOS. The development team includes: Walt Truszkowski, 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) for their major contributions.

12. References

[1] Havelund, K., Model Checking Java Programs using Java PathFinder. 1999. To appear in International Journal on Software Tools for Technology Transfer.

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

[3] Holzmann, H. J. 1991. Design and Validation of Computer Protocols. Prentice Hall Software Series.

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

[5] Truszkowski, W., Hallock, H. Agent Technology from a NASA Perspective. CIA-99, Third International Workshop on Cooperative Information Agents, Uppsala, Sweden, 31 July – 2 August 1999, Springer-Verlag.