Autonomous Real Time Requirements Tracing

George Plattsmier
Marshall Space Flight Center.
Huntsville, AL 35812
256-544-3963
George.l.Plattsmier@nasa.gov

Howard Stetson
Teledyne Brown Engineering
Huntsville, AL 35812
256-961-0399
Howard.K.Stetson@nasa.gov

Abstract – **One of the more challenging aspects of software development is the ability to verify and validate the functional software requirements dictated by the Software Requirements Specification (SRS) and the Software Detail Design (SDD). Insuring the software has achieved the intended requirements is the responsibility of the Software Quality team and the Software Test team. The utilization of Timeliner-TLX ™ Auto-Procedures for relocating ground operations positions to ISS automated on-board operations has begun the transition that would be required for manned deep space missions with minimal crew requirements. This transition also moves the auto-procedures from the procedure realm into the flight software arena and as such the operational requirements and testing will be more structured and rigorous. The auto-procedures would be required to meet NASA software standards as specified in the Software Safety Standard (NASA-STD-8719), the Software Engineering Requirements (NPR 7150), the Software Assurance Standard (NASA-STD-8739) and also the Human Rating Requirements (NPR-8705). The Autonomous Fluid Transfer System (AFTS) test-bed utilizes the Timeliner-TLX™ Language for development of autonomous command and control software. The Timeliner-TLX™ system has the unique feature of providing the current line of the statement in execution during real-time execution of the software. The feature of execution line number internal reporting unlocks the capability of monitoring the execution autonomously by use of a companion Timeliner-TLX™ sequence as the line number reporting is embedded inside the Timeliner-TLX™ execution engine. This negates I/O processing of this type data as the line number status of executing sequences is built-in as a function reference. This paper will outline the design and capabilities of the AFTS Autonomous Requirements Tracker, which traces and logs SRS requirements as they are being met during real-time execution of the targeted system. It is envisioned that real time requirements tracing will greatly assist the movement of auto-procedures to flight software enhancing the software assurance of auto-procedures and also their acceptance as reliable commanders.**

TABLE OF CONTENTS

1. INTRODUCTION

The Advanced Exploration Systems (AES) Autonomous Mission Operations (AMO) Project at Marshall Space Flight Center (MSFC) is researching the movement of ground operations to on-board operations for manned deep space missions. Command and control procedures are developed utilizing the Timeliner-TLX ™ auto-procedure system for autonomous fluid transfers within the AFTS test-bed located at MSFC. The research involves not only the development of intelligent auto-procedures but also the software standards and engineering requirements that would facilitate the qualification of auto-procedures as flight software. Auto-procedure analysis has shown that there are 4 types of procedures and that these have a consistent coding layout. Furthermore, a coding standard could be developed that could include the insertion of requirement identifiers within the coding layout, and that these identifier's could be paired to encompass the code that is developed for a specific requirement. The implementation allowed the team to develop a tracing program that would track auto-procedure execution in real-time, reporting the requirements that were encountered during the execution.

2. SOFTWARE REQUIREMENTS

All auto-procedure development should be driven by a Software Requirements Specification (SRS) which identifies the base requirements of the system being developed. It has been a standard practice at MSFC to label each requirement with a unique identifier. This unique identifier is then utilized by Software Quality personnel to track the requirements for test case analysis and during software testing to insure functional completeness. The

U.S. Government work not protected by U.S. copyright

Autonomous Fluid Transfer System (AFTS) SRS is utilized for deriving the list of unique identifiers for the complete system. The SRS was divided into 4 sections: General, Safety, Autonomous Operations, and Fault Detection Isolation and Recovery requirements. The General requirements encompassed the overall operation of the test-bed. The Safety requirements pertained to the safe operations and safety rules defined for the test-bed. The Autonomous Operations requirements specified the operations functions to be developed. The Autonomous Fault Detection Isolation and Recovery requirements specified the monitoring and real-time reaction to faults that were to occur. The Unique identifiers within the SRS provided uniqueness between requirement sections as shown in Figures 1 through 4.

3.1 GAFTS-0001 The software system shall query the crew to verify all manual valves are in the correct positions before operation of the AFTS.

3.2 GAFTS-0002 The software system shall allow transfer of fluid from the supply tank through the primary flow path to the multiuse tank.

3.3 GAFTS-0003 The software system shall allow transfer of fluid from the supply tank through the backup flow path to the multi-use tank.

3.4 GAFTS-0004 The software system shall allow the transfer of fluid from the multi-use tank through the return path to the supply tank.

3.5 GAFTS-0005 The software system shall provide for fluid heater control.

Figure 1 – AFTS General Requirements Sample

4.1 SAFTS-0001 The software system shall insure that fluid temperatures do not exceed 75 degrees F.

4.2 SAFTS-0002 The software system shall insure fluid heaters are safed whenever the fluid levels are below the fluid heater interface.

4.3 SAFTS-0003 The software system shall verify fluid transfer quantity requests from the crew are valid before initiating autonomous functions.

Justification: There has to be enough fluid to transfer and enough volume for the fluid to be received.

4.4 SAFTS-0004 The software system shall have the capability to safe the complete fluid transfer system with a single crew action.

Justification: In an emergency, there may not be enough time to execute a manual procedure

Figure 2 – AFTS Safety Requirements Sample

5.1 AAFTS-0001 The software system shall be capable of performing quarter tank fluid transfers over the primary flow path with a single crew action.

5.2 AAFTS-0002 The software system shall be capable of performing quarter tank fluid transfers over the backup flow path with a single crew action.

5.3 AAFTS-0003 The software system shall be capable of performing quarter tank fluid transfers over the return flow path with a single crew action.

5.4 AAFTS-0004 The software system shall be capable of performing half tank fluid transfers over the primary flow path with a single crew action.

5.5 AAFTS-0005 The software system shall be capable of performing half tank fluid transfers over the backup flow path with a single crew action.

5.6 AAFTS-0006 The software system shall be capable of performing half tank fluid transfers over the return flow path with a single crew action.

5.7 AAFTS-0007 The software system shall be capable of performing a crew selectable quantity tank fluid transfer over the primary flow path with a single crew action.

5.8 AAFTS-0008 The software system shall be capable of performing a crew selectable quantity tank fluid transfer over the backup flow path with a single crew action.

5.9 AAFTS-0009 The software system shall be capable of performing a crew selectable quantity tank fluid transfer over the return flow path with a single crew action.

5.10 AAFTS-0010 The software system shall be capable of performing a full tank fluid transfer over the primary flow path with a single crew action.

5.11 AAFTS-0011 The software system shall be capable of performing a full tank fluid transfer over the backup flow path with a single crew action.

5.12 AAFTS-0012 The software system shall be capable of performing a full tank fluid transfer over the return flow path with a single crew action.

5.13 AAFTS-0013 The software system shall be capable of controlling the temperature of the fluid.

5.14 AAFTS-0014 The software system shall message all autonomous actions performed during real-time.

Figure 3 - AFTS Autonomous Operations Requirements Sample

6.1 FAFTS-0001 The software system shall detect failures of the hardware during real-time operations.

6.2 FAFTS-0002 The software system shall safe hardware upon detection of failures during real-time operations.

6.3 FAFTS-0003 The software system shall recover failed operations during real-time operations.

6.4 FAFTS-0004 The software system shall message all autonomous actions performed during real-time operations.

Figure 4 - AFTS FDIR Requirements Sample

Notice that each SRS section has unique identifiers. As code is developed for the system, the unique identifiers are entered into the code via comment lines as identifier pairs that bound the code from top to bottom encompassing the requirement the code is implementing. This bounding by unique identifier pairs is essential in the compiler listing scan that is performed to obtain the line numbers that will be reported in telemetry during real-time execution.

3.TIMELINER-TLXCODING STANDARD

The unique requirement identifier inserted into the source code becomes part of the coding standard for Timeliner-TLX ™ auto-procedures. These identifiers must be in pairs and cannot be nested within a sequence. Figure 5 depicts the unique requirement identifier example placed within source code.

The example shows the GAFTS-0001 requirement for the manual valve status query to the crew. The requirement is paired at the beginning of the supported code and at the end. After compilation of the source code by the Timeliner-TLX ™ compiler, a compiler listing file is produced. The compilation produces a listing file where each executable line of code is assigned a unique line number as all code in the file is numbered contiguously. Figure 6 depicts the manual valve status query compilation listing that was shown first as source code.

```
--*** Now check that the test bed is in an operational state           ***
--*************************************************************************
-- GAFTS-0001 Manual Valve Status Query Requirement
confirm "HAL: Are the Manual Valves One and Two in the On Position?"
   when RESPONSE_RECEIVED WITHIN 1:00 then -- crew one minute to respond
      if OPERATOR_RESPONSE = AFFIRMATIVE THEN
         MESSAGE "HAL: AFTS Test Bed is Ready for Operations!"
         Set ReadyForOps = TRUE
      else
         WARNING "HAL: AFTS Test Bed is Not Ready for Operations!"
         Set ReadyForOps = FALSE
      end if
   otherwise
      disregard "HAL: Manual Valve Inquiry timeout!"
      WARNING "HAL: Automatic Bundle Installation Inhibited "
      Set ReadyForOps = FALSE
   end when
-- GAFTS-0001 Manual Valve Status Query Requirement
```

Figure 5 – Unique Identifier Pairs

```
      --*** Now check that the test bed is in an operational state           ***
      --*************************************************************************
      -- GAFTS-0001 Manual Valve Status Query Requirement
25            confirm "HAL: Are the Manual Valves One and Two in the On Position?"
26            when RESPONSE_RECEIVED WITHIN 1:00 then -- crew one minute to respond
27                if OPERATOR_RESPONSE = AFFIRMATIVE THEN
28                    MESSAGE "HAL: AFTS Test Bed is Ready for Operations!"
29                    Set ReadyForOps = TRUE
30                else
31                    WARNING "HAL: AFTS Test Bed is Not Ready for Operations!"
32                    Set ReadyForOps = FALSE
33                end if
34            otherwise
35                disregard "HAL: Manual Valve Inquiry timeout!"
36                WARNING "HAL: Automatic Bundle Installation Inhibited "
37                Set ReadyForOps = FALSE
38            end when
      -- GAFTS-0001 Manual Valve Status Query Requirement
```

Figure 6 – Compiler Listing Example

The GAFTS-0001 requirement is bounded by line numbers 25 through 38 which would be reported in real-time during execution of this segment of code. There is no limit to the number of lines of code within a single requirement nor is there a limit to the number of requirements.

4. SRS, TIMELINER PARSER

A program written in ANSI C was developed to automatically generate the Autonomous Real Time Tracking Sequence. The storage mechanism employed in this program is a single linked list data structure. A single linked list structure consists of a head pointer which points to the entire list by storing a pointer to the first node. Each consecutive node contains a pointer to the next node and so on. The last node has its next field set to NULL to mark the end of the linked list. A single linked list diagram is depicted in Figure 7.

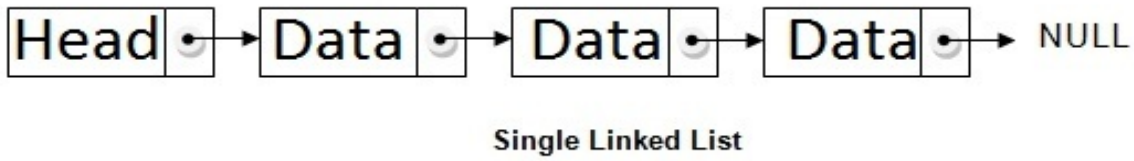

Figure 7 – Single Linked List Diagram

The AFTS SRS and Timeliner TLX™ compiler listing files are given as inputs to the program. The program begins by reading in the AFTS SRS saved in a .txt extension. The parsing of the SRS document consists of storing all of the requirements found in the SRS into the linked list structure. The sample code segment in Figure 8 depicts the insertList function which inserts the requirement into the linked list.

Parsing of the Timeliner-TLX™ files involves first checking to make sure the requirement found matches a requirement in the SRS. This check is accomplished using a search function which iterates through the linked list of requirements. An example of the SRS to Timeliner-TLX™ requirement validation is depicted in figure 9.

```
// Define a linked list structure to store the SRS requirement
typedef struct requirement
{
        char req[10]; //allocate a string array of ten elements
        struct requirement *next; //next pointer to point to next element in the linked list
}srs_req;

if(strstr(str,"%")) //check if substring contains Requirement unique identifier in SRS
{

        if(p = strchr(str, '%')) //find first occurrence of % in substring
                {
                strncpy(&temp[0], p, 11); //copy from eleventh element to temp array starting at element zero
                temp[10] = '\0'; //Null terminate the string
                insertList(temp);  // Insert requirement into linked list
                }
}
```

Figure 8 Sample linked list implementation

```
if(strstr(str,"%")) //check if substring contains unique Requirement identifier compiler listing file
{
	if(p = strchr(str, '%')) //find first occurrence of % in substring
	{
	strncpy(&temp[0], p, 11); //copy from element eleven to zero element of temp array
	temp[10] = '\0'; //null terminate temp array
	strcpy(temp2,temp); //copy contents of temp array to temp2 array
		if(strcmp(temp1,temp2)==0) //check to make sure the requirement matches
		{
			sscanf(str + 19, "%[^\n]", test4); //copy requirement description
			test4[ strlen(test4) - 1 ] = '\0'; //null terminate test4 array

			if(searchList(temp1, temp2)) //search linked list for existing SRS requirement
			{
				result = beg_count + end_count; //store line number
				genBundle(test2,temp1,test3,test4,beg_count,result); //Generate Requirement Tracker file
				end_count = 0;//clear end count variable
				first = false; //reset second requirement pair search loop
				done = true; //reset while loop search flag
			}
			else
			{
				printf("Requirement %s%s ",temp1, " not found in SRS\n");//Inconsistent requirement
				result = beg_count + end_count; //store line number
				genBundle(test2,temp1,test3,test4,beg_count,result); //Generate Requirement Tracker file
				end_count = 0;//clear end count variable
				first = false; //reset second requirement pair search loop
				done = true; //reset while loop search flag
			}
		}
		else
		{
			printf("breaking loop, second match not found\n");//Inconsistent requirement pairing
			done = true; //reset while loop search flag
		}
	}
}
//---------------------------------------------
// Function: searchList(char *req, char *retVal)
// Purpose: Search for a requirement in the
// linked list, and return result into the variable
// pointed to by *retVal.
// Returns: TRUE if search was successful
// or FALSE if the search failed.
//---------------------------------------------
char searchList(char *req, char *retVal)
{
	ListItem *temp;

	temp = head; //set temp equal to beginning of linked list
	while((temp != NULL) && (!strum(req,temp->req)==0)) //Continue while the list is not empty
	{
		temp = temp->next; //advance to next node in linked list
	}
	// If item not found or list is empty return FALSE
	if(temp == NULL) return false; //requirement not found, so return false
	else
		strcpy(retVal,temp->req); //copy requirement found into variable

	return true;// Signal successful search
}
```

Figure 9 Sample search function implementation

A key point to notice in Figure 9 is that the program will catch if the Timeliner TLX™ source code and SRS requirements do not match. Also, the program will determine if the Timeliner TLX™ unique identifiers are not paired, or have an erroneous or misspelled unique identifier. The program will conclude with generating a Timeliner TLX™ source file which will be converted into the Tracker Sequence.

5. THE TRACKER SEQUENCE

The Tracker Sequence is the output file of the SRS/Timeliner-TLX™ file parser. This file will be compiled which generates an executable and a listing file. The executable is then installed into the Timeliner-TLX™ Executor / Engine. Upon installation, the operator must start the Tracker Sequence. As soon as the Tracker Sequence becomes active, this sequence will run every second scanning for active bundles. The first check in the sequence will determine if a given bundle is active. The logic will then decide if the specified sequence in the current bundle is active. If the sequence is active, it will then check if the sequence statement is within a range of line numbers. If the active sequence lies between the sequence statement range, a message will then be logged to the Timeliner-TLX™ engine log file indicating a requirement has been encountered. Figure 10 depicts a Sequence Tracker code segment.

```
Sequence TRACKER Active--***
--*** We start our control loop to monitor every second
--***
Every 1.0 then
 If AWTS_HAL_MAIN.BUNSTAT = BUN_ACTIVE Then -- Is the bundle active?
   If AWTS_HAL_MAIN.Initialize.SEQSTAT = SEQ_ACTIVE Then -- Is the Initialize Sequence active?
     If AWTS_HAL_MAIN.Initialize.SEQSTMT IN 25..38 then -- Current line within the req range?
       Message "GAFTS-0001 Manual Valve Status Query Requirement"
     End If
     If AWTS_HAL_MAIN.Initialize.SEQSTMT IN 39..54 then -- Current line within the req range?
       Message "GAFTS-0006 Autonomous Procedure Installation Requirement"
     End If
   End If
 End If
```

Figure 10 – Example Tracker Code

6. REAL TIME EXECUTION

The Tracker Sequence must be started first in the execution order as installing the HAL_Main Bundle will begin autonomous operations for the AFTS Test-Bed. The Tracker Sequence monitors for the bundle installation of HAL_Main. When the bundle is installed, sequences within the bundle become "active" automatically. Once the HAL_Main bundle installation is detected, the Tracker Sequence monitors for the individual sequences to become active. Once the sequences become active, Timeliner-TLX ™ begins reporting execution line numbers. The Tracker Sequence monitors the reported line numbers for requirement ranges, matching a line number to the requirement line number ranges. When a match is found, a message statement is issued to the Timeliner-TLX™ console. A log message is written to the log file. This same interaction is performed for each bundle and sequence that is installed and becomes "active", as many times as the sequence is executed. After the test-plan is executed, the log is utilized for analysis of requirement coverage. For the AFTS Test-Bed, the log depicts the time of execution and the order of the test-plan as executed in real-time. If multiple test plans are to be executed, such as an autonomous plan of fluid transfers, the Tracker sequence can simply stay active as it continually scans for bundle installations and active sequences until manually stopped by the operator. There are occasions where a requirement is satisfied by only a few lines of code producing a line number range that is small. The AFTS Test-Bed had 1 requirement (validate transfer quantity), that encompassed only one or two lines of code. During testing, the team noticed the validation requirement was not being reported, although the validation code was executed. This meant that the Timeliner-TLX ™ execution engine was executing these lines of code rapidly enough that the reported execution line was past the requirement bounded code. The TLX engine executes at a 1 Hz rate within the CPU, and the number of statements that the engine can execute is dependent upon the number of sequences that are currently active. The more sequences that are active, the less number of statements within each sequence are executed. The solution to force reporting of bounded statements is to place a "Wait 1" statement within the requirement range. The WAIT statement provides a one second wait in execution and will force the sequence to give up its execution time slice within the TLX engine, and the current line number will be reported.

7. LOG OUTPUT AND ANALYSIS

Figure 11 shows the log output produced by the Tracker Sequence. The time frame of this log file shows the initialization of the AFTS through three autonomous fluid transfers; one via the back-up leg with one of a quarter tank quantity, a second transfer where the crew selects the quantity transferred over the backup leg, and the third transfer which is a crew selectable quantity transferred over the primary leg. During initialization of the AFTS, the software system queries the crew on the status of manual valves which does not provide telemetry. If the manual valves are not in an operational state, or the query is not answered within one minute, the system inhibits the autonomous installation of auto-procedures, preventing operation of the test bed in a non-operational state. The log traces the date and time, the requirement encountered and executed, the Timeliner-TLX™ Bundle Name that contained the requirement, the tracking tag of the bundle which is created at compilation time and utilized for configuration management of the software system, and the requirement ID including the text that was encountered during execution. The Tracking Tag is another unique feature of the Timeliner-TLX ™ system as this internal identifier is also reported in telemetry during execution allowing operations to uniquely identify the procedures in execution. The Tracking Tag can be decoded as it is a date/time stamp, produced in binary coded decimal, after a successful compilation of the procedure. The log output shows that after the crew manual valve query, autonomous installation of the AFTS software occurred and that three fluid transfers of different types were directed by the crew. The log is then used to determine which requirements were met during the current test plan. In this way, test plans can be tailored to include operations that were missed. Software Quality personnel can now scan the log for requirement and test plan analysis.

TIME TAG	BUNDLE NAME	TRACKING TAG	MESSAGE TEXT
07/16/13 09:22:18	REQUIREMENT_TRACER2	1307160850040151	GAFTS-0001 Manual Valve Status Query Requirement
07/16/13 09:22:24	REQUIREMENT_TRACER2	1307160850040151	GAFTS-0006 Autonomous Procedure Installation Requirement
07/16/13 09:22:26	REQUIREMENT_TRACER2	1307160850040151	GAFTS-0006 Autonomous Procedure Installation Requirement
07/16/13 09:22:27	REQUIREMENT_TRACER2	1307160850040151	GAFTS-0006 Autonomous Procedure Installation Requirement
07/16/13 09:27:58	REQUIREMENT_TRACER2	1307160850040151	AAFTS-0002 Quarter Tank Backup Transfer Requirement
07/16/13 09:38:10	REQUIREMENT_TRACER2	1307160850040151	AAFTS-0008 Crew Selectable Backup Transfer Requirement
07/16/13 09:39:37	REQUIREMENT_TRACER2	1307160850040151	AAFTS-0007 Crew Selectable Quantity Primary Transfer Requirement

Figure 11 – Example Log Output

8. SUMMARY

The Command and control procedures developed utilizing the Timeliner-TLX ™ within AFTS test-bed have been instrumental in proving the concept of automated and autonomous operations for a deep space mission. The testing has helped to pave the path for extended research and validating the transition from auto-procedure to flight software. The idea of moving command and control from ground control centers to the crew on a manned deep space mission involves extended research of how to develop intelligent auto-procedures and how much of the operational environment will change. The early analysis of Auto-procedures has shown that there are 4 types of procedures. Extensive testing and research will be needed to solidify this confidence and exactly detail the development requirements for auto-procedures to be an accepted norm for operations. Further hardware software integration testing will lead to a more detailed concept of what an autonomous architecture will entail and how to qualify the software for flight. The Timeliner-TLX ™ system's unique line reporting feature provides a process to verify and validate what requirements are met in real time to aid hardware and software designers in this endeavor. It is envisioned that on-board intelligent auto-procedures will be required on manned deep space missions due to the extreme communications delays, and auto-procedure updates will still be occurring with Earth-based assets. A rapid verification and validation of auto-procedures will be needed for such "on mission" vehicle updates.

REFERENCES

[1] Stetson, H.K.; Haddock, A.T.; "Automated Operations Development for Advanced Exploration Systems," American Institute of Aeronautics and Astronautics, Space Ops 2012, Stockholm Sweden

BIOGRAPHY

George Plattsmier *is employed by NASA'S Marshall Space Flight Center (MSFC) in Huntsville Alabama, in the Mission Operations Lab (MOL). Currently, Mr. Plattsmier is a member of the Simulation and Computer System Team with NASA'S Advanced Exploration Systems-Autonomous Mission Operation (AES-AMO) project. Prior to the AES-AMO project, he was a team member of a flight software tools team which developed a requirements traceability tool. He holds a Bachelor of Science degree in Computer Engineering from the University of Alabama in Huntsville. Mr. Plattsmier, and his wife Andrea have three children; Morgan, Louis, & Liam.*

Howard K. Stetson *is a contractor for Marshall Space Flight Center, Space Systems Operations and is currently working as an analyst for the Advanced Exploration Systems-Autonomous Mission Operations project and has over 34 years of experience in software development and engineering, encompassing numeric intensive computing, parallel processing, computer graphics, simulation and modeling, computational fluid dynamics, real-time C&C operations, operations automation, and software integration and test. Preceding the AES projects, Mr. Stetson designed, developed and implemented the Higher Active Logic (HAL) autonomous system for ISS payloads and has worked flight operations development for the Ares and SLS programs. Mr. Stetson is also a member of and instructor for the United States Parachute Association and has over 3239 jumps to date.*