Agent SMITH:
Evolution of a Test Tool in Tcl/Tk

John J. Seal

Abstract

This paper describes the evolution of the Story Maker Interface Test Harness (SMITH), a test tool developed by Raytheon Technical Services Company (RTSC) for internal use on one of its projects. Development started with a sketch made by a project engineer, and proceeded in four incremental stages: First, implement the sketch as proof of concept; second, automate the tool using test case files; third, develop a C extension to provide hooks into the target system; and fourth, analyze the target system's response. These stages tracked the changing needs of the project, from development of the interface code before real hardware was available, through unit test, and finally into full-scale system integration and qualification testing.

Summary

RTSC's Customized Engineering and Depot Support (CEDS) site in Indianapolis, IN, is responsible for developing and maintaining a system known as Story Teller. Our customer asked us to integrate a new system, called Story Maker, to be supplied by a 3rd party. We had the Interface Control Document (ICD), but actual hardware would not be available for quite some time. Instead of just coding blindly to the ICD and hoping for a successful "big bang" integration, we decided to develop a tool to exercise the interface until real hardware was available.

The lead engineer for Story Teller used Microsoft Word (?) to sketch out a notional Graphical User Interface (GUI) for that purpose, and marked it up with notes about what the various elements should do. [Include sketch.] For comparison, the final SMITH tool is shown beside it. [Include screenshots.] The final tool matches the initial concept very closely, but there are some significant differences, too.

The first phase of development was to simply implement the notional sketch almost literally, as a proof of concept. The ICD defined a simple TCP/IP network protocol that was easily implemented. The initial GUI allowed the user to construct a Maker message, send it to Teller, and see the response. The initial concept included fields to describe the expected response in narrative form, but we quickly realized that by adding fields to describe the expected response in detail, the tool could automatically check the actual response for correctness. Thus, the final tool has three main columns instead of two.

Once the proof of concept was demonstrated, the next step was to add data-entry aids. Story Teller has data files that define the valid values for certain fields, and we use those same files to create pull-down menus (indicated by downward-pointing triangles) for the entry fields. When the user selects a value for one field, it changes the values available for the others; this makes it easy to specify valid and consistent test cases. The user can type directly into the fields to create invalid or inconsistent test cases.

The second phase was to automate the tool by allowing it to read test cases from a file. This was important because, at that time, the chief role envisioned for the tool was for Unit Test of the interface code. Just verifying that the code followed the ICD required several hundred test cases, and we didn't want the user to have to enter them manually! This added "transport controls" to the GUI so the test case file could be stepped through, played automatically, rewound, etc. Progress and results were shown in a text pane at the bottom of the GUI, with discrepancies between actual and expected results highlighted.

The initial implementation required that the user prepare the test case file in a text editor. We developed a fairly free format and a parser to support it, but the testers (who are not programmers) didn't like it. They pointed out the blindingly obvious: There's already a GUI to specify an individual test case, so add a button to append it to the test case file! In that way the testers could use the GUI to build the test case files, experimenting with each single test case until they got it just the way they wanted it, then saving it. Bringing the tool to this point took about two weeks of real time. The tool in this form was used extensively by both developers and testers.

Once the new interface was working well, the Teller engineer started using the tool as if it were a Maker simulator, that is, to exercise Teller instead of just the interface. He pointed out that valid commands would cause Teller to send certain messages to other systems, and other systems to send certain messages to both Teller and Maker. Would it be possible for SMITH to verify Teller's external response to Maker messages, rather than just verifying that they were correctly accepted or rejected?

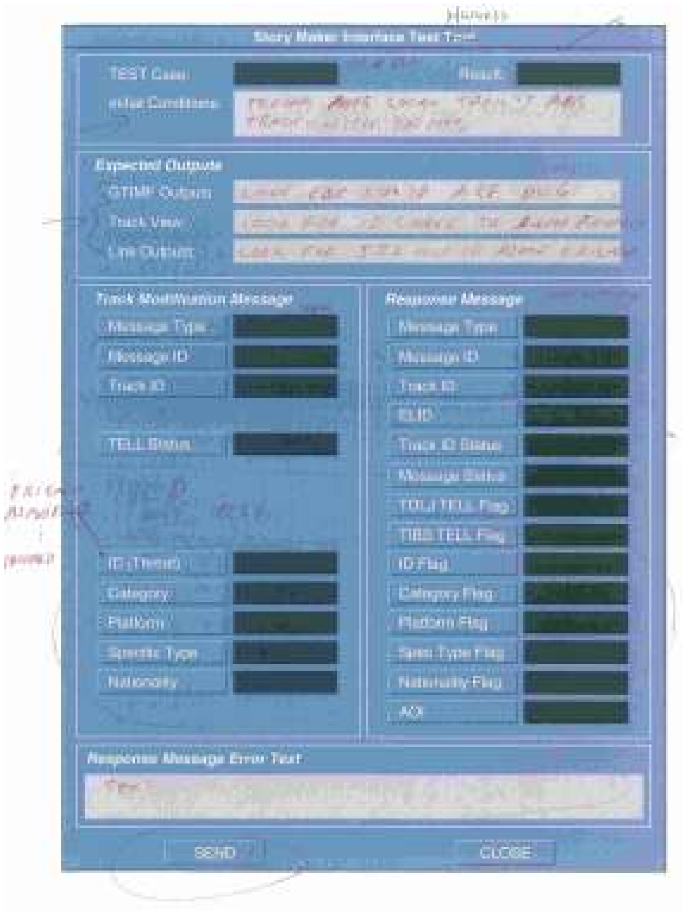
The third phase of development was adding hooks into Teller so SMITH could verify the external message traffic. This involved having SMITH open a server socket so it could receive messages intended for Maker, and registering to receive messages sent and received by Teller. Teller uses a peculiar homegrown Inter-Process Communication (IPC) scheme, so it was necessary to write a simple C extension to wrap the key functions and export them to Tcl. Could I have used CriTcl or FFIDL instead? I don't know, but compiling a C extension to a shared library is simple.

At this point the tool could not only verify that valid messages were accepted, but that Teller responded correctly, too. The user could choose to log external message traffic as simple events or hex/ASCII dumps. The external messages are in Generic Tactical Information Message Format (GTIMF). Teller contains another tool, written in C, that can decode them to "human readable" form, but it's very difficult to use both tools together and correlate the results.

The fourth phase was to parse the external GTIMF messages into true human readable form. (The Teller tool parsed the fields and presented them numerically, with no interpretation.) The GTIMF specification documents the meaning of each field, so we wrote a parser to display fully-interpreted messages. The specification of the GTIMF protocol in Tcl is very flexible and could be easily extended to other formats. It took less than a week of real time to design, code, test, and document the GTIMF parser!

The final SMITH tool has been very well received and widely used. System engineers use it to explore various “what if?” scenarios. Developers use it to debug other Teller changes being proposed or implemented. Testers use it to provide repeatable stimulus during system integration and qualification tests.

These pictures are UNCLASSIFIED.



Story Maker Interface Test Tool

TEST Case:		Result:	
Initial Conditions:	[illegible]		

Expected Outputs

GTMP Outputs	[illegible]
Track View	[illegible]
Link Outputs	[illegible]

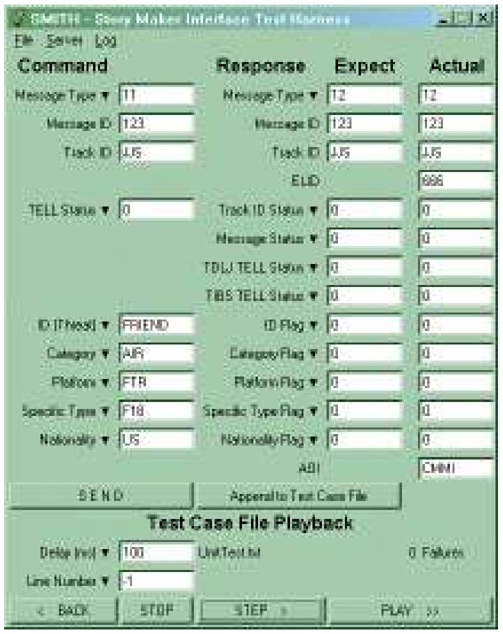
Track Modification Message

Message Type
Message ID
Track ID
TELL Status
ID (Threat)
Category
Platform
Specific Type
Nationality

Response Message

Message Type
Message ID
Track ID
ELID
Track ID Status
Message Status
TDL/J TELL Flag
TIBS TELL Flag
ID Flag
Category Flag
Platform Flag
Spec Type Flag
Nationality Flag
AOI

Response Message Error Text

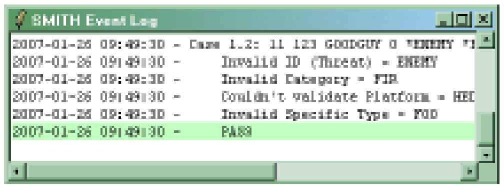


SMITH - Story Maker Interface Test Harness

Command		Response	Expect	Actual
Message Type ▼	11	Message Type ▼	12	12
Message ID	123	Message ID	123	123
Track ID	JJS	Track ID	JJS	JJS
		ELID		666
TELL Status ▼	0	Track ID Status ▼	0	0
		Message Status ▼	0	0
		TDL/J TELL Status ▼	0	0
		TIBS TELL Status ▼	0	0
ID (Threat) ▼	FRIEND	ID Flag ▼	0	0
Category ▼	AIR	Category Flag ▼	0	0
Platform ▼	FTR	Platform Flag ▼	0	0
Specific Type ▼	F18	Specific Type Flag ▼	0	0
Nationality ▼	US	Nationality Flag ▼	0	0
		AOI		CHMI

Test Case File Playback

Delay (ms) ▼	100	UnitTest.txt	0 Failures
Line Number ▼	-1		



SMITH Event Log

```
2007-01-26 09:49:30 - Case 1.2: 11 123 GOODGUY 0 "ENEMY "1
2007-01-26 09:49:30 -     Invalid ID (Threat) = ENEMY
2007-01-26 09:49:30 -     Invalid Category = FIR
2007-01-26 09:49:30 -     Couldn't validate Platform = HED
2007-01-26 09:49:30 -     Invalid Specific Type = F00
2007-01-26 09:49:30 -     PASS
```