Visualization Tools for Validating Software of Autonomous Spacecraft

Reid Simmons and Gregory Whelan

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
(reids@cs.cmu.edu, whelan@cs.cmu.edu)

ABSTRACT

Spacecraft autonomy is becoming an increasingly important technology. Yet the very nature of autonomy – on-board decision making and largely unattended operation – makes it important that such systems be thoroughly tested and validated. We are developing software visualization tools to assist in the validation process. The tools, which are designed to facilitate human problem solving, combine graphical layout and color to present "gestalt" views of system execution, together with interactive facilities for browsing, searching, and tracking down potential problems. This paper describes two tools being developed in conjunction with the NASA New Millennium Program – one for visualizing inter-process comm-unication and one for visualizing plan execution.

1. INTRODUCTION

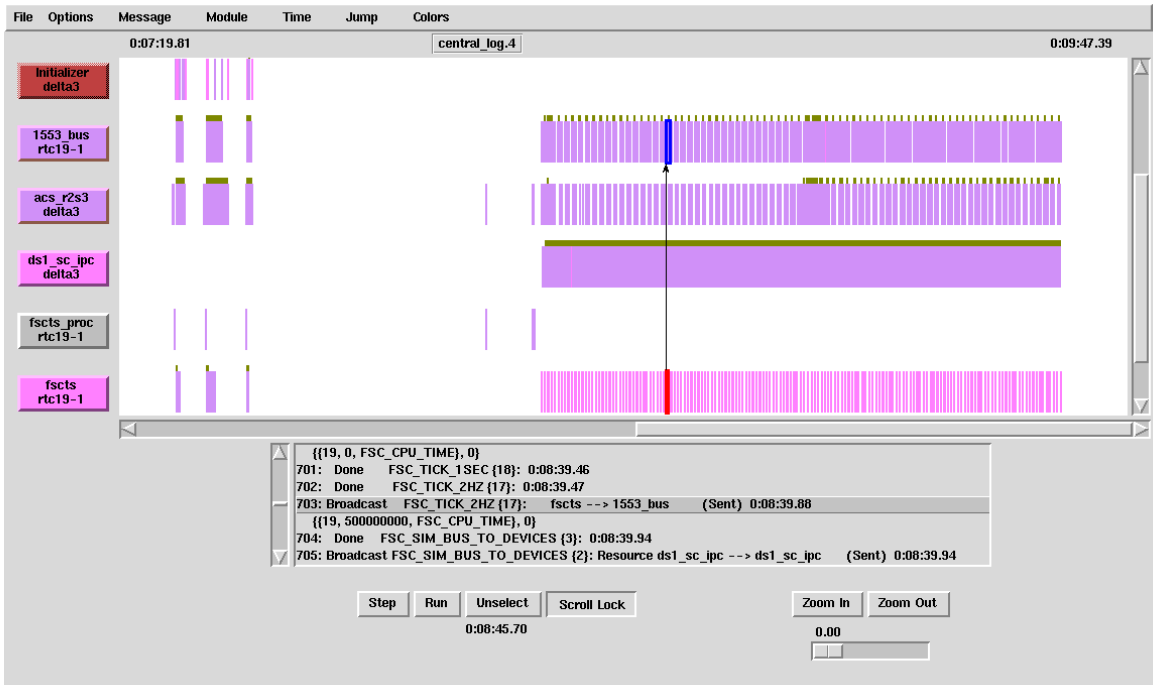
To meet the needs of future inter-planetary and near-earth space exploration, designers are increasingly interested in making spacecraft more autonomous. The rationale for this includes enabling more complex missions, such as small body encounters and landings and reducing ground operations cost by making decisions on board. Creating highly autonomous spacecraft is a high priority of the NASA New Millennium Program, which seeks to flight validate cutting-edge technologies. In particular, the autonomous *Remote Agent* [2] is being developed as part of the Deep Space One mission (an asteroid and comet flyby), anticipated to launch in July 1998.

Typically, such spacecraft use a concurrent, distributed software architecture that coordinates actions and exchanges information via message passing [9]. From a software-engineering perspective, such an architecture has great advantages in terms of independent development of modules, data abstraction and data hiding, and software reuse. Validating and debugging such large, concurrent systems can be a nightmare, however. For example, subtle flaws in the design of module interfaces often manifest themselves only during system integration.

In the case of autonomous spacecraft, a critical validation test is whether the system responds to stimuli (both internal and external) appropriately, and in time. The system can fail due to faulty algorithms (leading to inappropriate or missing responses) or due to limited computational resources (leading to delays and bottlenecks). Such problems can manifest themselves at various levels of abstraction – from low-level servo control, through high planning and plan execution.

To aid software developers in the New Millennium Program, we have created tools for visualizing the execution of such autonomous, concurrent systems. The tools parse log files produced during system execution and present the data in intuitively understandable formats. One tool, called *comview*, displays patterns of inter-process communication. It can help answer questions such as when messages are sent and to whom, and where bottlenecks are occurring. Another tool, called *planview*, visualizes the execution of plans (command sequences). It propagates temporal constraints between plan segments in order to detect constraint violations that signal potential plan failures.

Unlike other visualization tools for concurrent systems that focus primarily on automated analysis [4, 5, 6], *comview* and *planview* are geared towards human analysis. This influenced several design decisions. First, emphasis was placed on graphical layout and use of color. We felt that the overall visual structure should facilitate a "gestalt" view of system execution, in order to aid users in homing in on a given problem. Second, great emphasis has been placed on the interactivity of the tools. We felt that the ability to flexibly browse, search, and trace through patterns of system execution is important in the detailed work needed to pinpoint the source of problems. Finally, as we become more familiar with how these tools are used in practice, we intend to focus on automating the detection of common types of problems.



```
   {{19, 0, FSC_CPU_TIME}, 0}
701:   Done      FSC_TICK_1SEC {18}:  0:08:39.46
702:   Done      FSC_TICK_2HZ {17}:  0:08:39.47
703: Broadcast   FSC_TICK_2HZ {17}:     fscts --> 1553_bus       (Sent)  0:08:39.88
   {{19, 500000000, FSC_CPU_TIME}, 0}
704:   Done    FSC_SIM_BUS_TO_DEVICES {3}:  0:08:39.94
705: Broadcast FSC_SIM_BUS_TO_DEVICES {2}: Resource ds1_sc_ipc --> ds1_sc_ipc     (Sent)  0:08:39.94
```

Figure 1. The *comview* Tool

2. Visualizing Message Traffic

While careful up-front interface design can help simplify system integration, our experience has been that integration remains one of the key problems in developing and validating autonomous systems. Often, due to the complex system interactions, problems can be detected only by running the system and analyzing the patterns of inter-process message traffic. Important validation questions with respect to message traffic include determining where bottlenecks occur and determining whether modules are responding to their inputs appropriately.

The *comview* tool works with a message passing substrate, adapted from the Task Control Architecture [10], that provides both publish/subscribe (broadcast) and client/server (query) type messages. The inter-process communications package can log message traffic, indicating source and destination modules, when the message was sent, the data sent with the message, how long it was queued (if at all), and how long the destination module took to handle the message. *comview* operates by parsing the (human readable) log files and displaying them in various ways. The logs files are typically processed after a run is completed, but *comview* can also run in real time, parsing the log file as it is being generated.

For portability, maintainability, and ease of development, *comview* (and *planview*) are implemented using standard software packages. The graphical user interface and the interactive facilities are implemented with Tcl/Tk library [8]. The log file parser is written using lex and yacc [7]. Only a relatively small portion of the tools needed to be written directly in C.

As mentioned, we feel it is important to present a “gestalt” view of the message traffic to facilitate finding potential trouble spots. *comview* does this primarily through its graphical layout, which takes advantage of the human capacity for processing spatial information. Information is laid out in a Gantt chart format (Figure 1), where each module (process or task) is displayed on a separate row, and each rectangle on a row indicates that a message was received by that module (the width of the rectangle being proportional to the time spent by the module in handling the message). The message rectangles are color-coded to indicate message status (e.g., whether the module is sending or receiving messages, or waiting for a response), and thin orange bars above the message rectangles indicate when messages are being queued (while the corresponding module is busy handling other messages).

This layout makes it easy to spot modules that are over (or under) utilized, and to spot where bottlenecks occur (indicated by queued messages, as illustrated by the **ds1_sc_ipc** module in Figure 1). Also, if there are regular patterns to the messages, as is often the

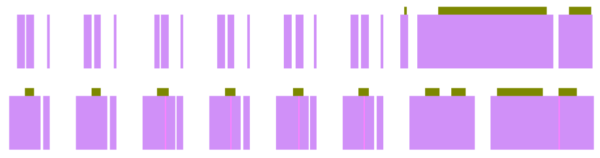
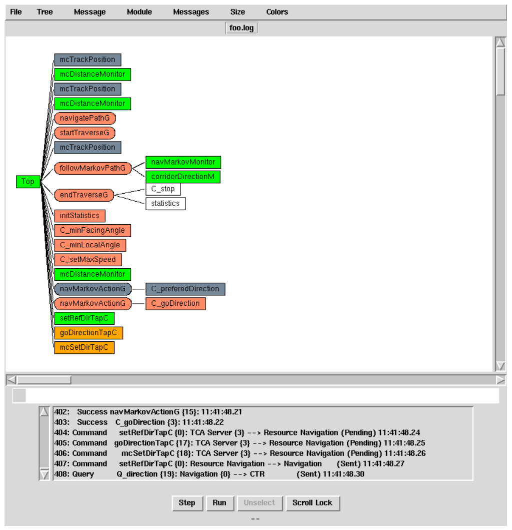


Figure 2. An Anomaly in a Pattern of Messages



```
402:   Success navMarkovActionG {15}: 11:41:48.21
403:   Success   C_goDirection {3}: 11:41:48.22
404: Command    setRefDirTapC {0}: TCA Server {3} --> Resource Navigation (Pending) 11:41:48.24
405: Command  goDirectionTapC {17}: TCA Server {3} --> Resource Navigation (Pending) 11:41:48.25
406: Command     mcSetDirTapC {18}: TCA Server {3} --> Resource Navigation (Pending) 11:41:48.26
407: Command    setRefDirTapC {0}: Resource Navigation --> Navigation     (Sent) 11:41:48.27
408: Query        Q_direction {19}: Navigation {0} --> CTR            (Sent) 11:41:48.30
```

Figure 3. A Hierarchical View

case, then anomalies tend to stand out. For instance, in Figure 2, the top module begins with a regular pattern of two short messages followed by one very short message, but then the pattern changes abruptly in the middle of the run.

Another useful way to view message flow is hierarchically – when a module receives a message, it often sends additional messages in response, which in turn get handled and cause other messages to be sent, etc. With information viewed this way, as an invocation tree (Figure 3), other types of message flow patterns become visually apparent, again making it easy to spot deviations from regular patterns.

While such "gestalt" views are useful for quickly spotting the general areas where problems exist, it still usually takes a bit of detective work to pinpoint the exact cause of a problem. In particular, the root cause may lie somewhere along a chain of messages, and the validation problem is to determine exactly where the message flow differs from expectations.

To facilitate this mode of problem solving, *comview* includes many facilities for interactively examining message flow. For example, when a message rectangle is selected (Figure 4), *comview* displays the flow of communication graphically (an

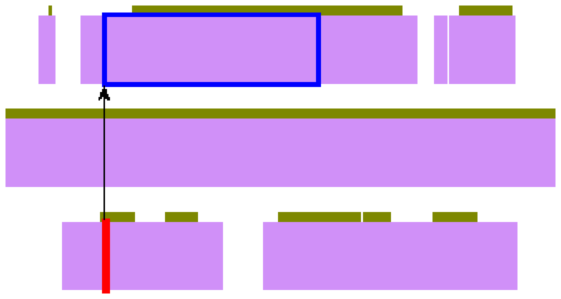


Figure 4. Connection From Sender to Receiver

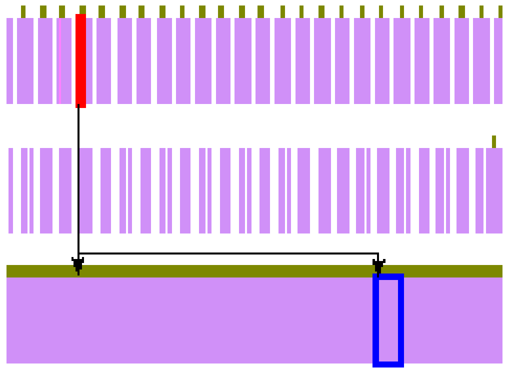
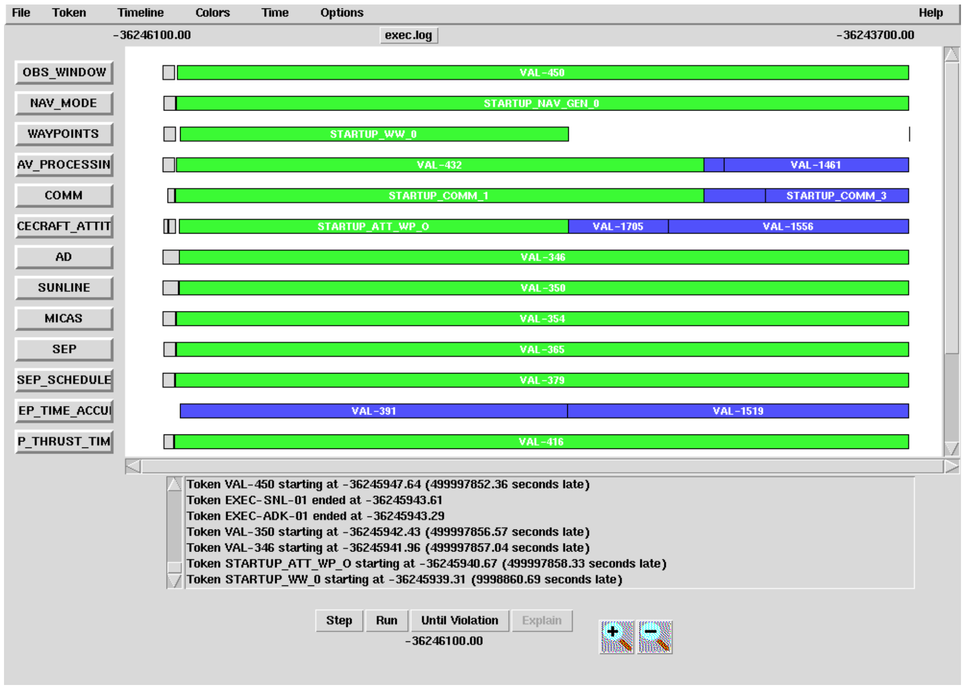


Figure 5. Visualizing a Queued Message

arrow from sender to receiver), as well as textually (the corresponding line in the log file is highlighted). If a message is queued, one sees graphically when it was first sent and then when it was eventually handled (Figure 5). One can search for a message by name, by source or destination module, by time of the message, even by the content of the message data itself. Other useful interactive features of *comview* include the ability to rearrange rows, to ignore displaying subsets of messages, to zoom and scroll, and to change the mappings between message status and colors.

We now present two examples where *comview* was used to help solve problems in the Deep Space One software system. The symptom in both cases was an unstable control loop, caused by a module making decisions based on old data. While the problems were clearly caused by delays somewhere, the problem was to figure out where (and why). The first case was largely solved with the "gestalt" view: A cursory examination of the display showed that one module had a large number of queued messages (Figure 1). By highlighting messages handled by that module (Figure 5), it became apparent that the queueing (delay) time was increasing with each successive message. A closer examination of the individual messages confirmed that new messages were being generated faster than it took the module to handle those types of messages. While, in theory, this could have been caused by a bug in either the sender (generating too fast) or the receiver (handling too slowly), in this case it turned out to be a problem with the receiver.

The other example illustrates the use of the browsing facilities. Again, the symptom was a delay in propagating current data. In this case, the "gestalt" view revealed little. Instead, we had to trace the flow of individual messages back through to their original sources. As a result of doing this for several instances of a key message type, we began to infer a pattern. We also noted that the pattern was broken in several instances. It turned out (by bringing in the developer



```
Token VAL-450 starting at -36245947.64 (499997852.36 seconds late)
Token EXEC-SNL-01 ended at -36245943.61
Token EXEC-ADK-01 ended at -36245943.29
Token VAL-350 starting at -36245942.43 (499997856.57 seconds late)
Token VAL-346 starting at -36245941.96 (499997857.04 seconds late)
Token STARTUP_ATT_WP_O starting at -36245940.67 (499997858.33 seconds late)
Token STARTUP_WW_0 starting at -36245939.31 (9998860.69 seconds late)
```

Figure 6. The *planview* Tool

of the module in question) that that module was keeping an *internal* queue of data, and that it was not flushing the queue, but rather sending out only the first (oldest) element each cycle. The developer had assumed that only one new datum would be received each cycle. Using *comview*, we were easily able to document times when that assumption was violated.

An obvious extension to *comview* would be the ability to specify such types of message patterns and to detect them (or, more specifically, to detect *violations* of the patterns) automatically. We are currently starting to address this, and intend to integrate it into the *comview* framework.

3. VISUALIZING PLAN EXECUTION

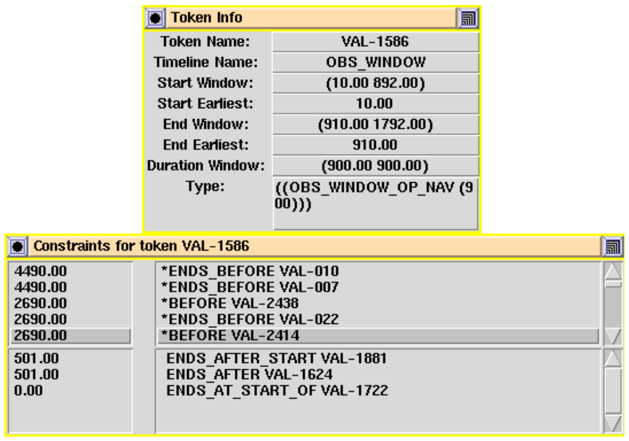
Nowadays, many autonomous systems are designed with a three-layer architecture, consisting of a behavioral/real-time layer, an executive/sequencing layer, and a planning layer [3, 9]. The executive layer is responsible for executing plans produced by the planner, monitoring plan execution, and recovering from exceptional situations. As such, validating this component is essential to ensure the successful operation of the overall system.

One important aspect in validating an executive is demonstrating that it executes plan segments at the appropriate time and in the appropriate sequence. The *planview* tool is designed to provide that capability for the New Millennium *Remote Agent* executive. This executive layer uses a plan representation based on timelines and tokens [9]. A *timeline* represents the evolution over time of a state variable of the system (e.g., the state of the main engine). Each timeline consists of a contiguous sequence of *tokens*, where a token represents the value of the state variable over some time interval (e.g., the main engine is thrusting). A token has an expected duration, and start and end time timepoints. Uncertainty in the duration and the time point windows is represented using ranges of time values. In addition, tokens may have temporal constraints between them (e.g., token **VAL-1586** must end before token **VAL-2414** begins; token **VAL-1586** must end after token **VAL-1881** starts).

The task of the *Remote Agent* executive is to achieve the state values associated with the tokens, while respecting their temporal constraints and time windows. Faults occur when the executive cannot achieve a token within its specified temporal window. The task of the *planview* tool is to detect violated constraints and help users track down the root causes of those faults.

As with *comview*, *planview* operates by parsing log files. The log files, produced by the executive, contain a description of the plan being executed (timelines, tokens, and constraints) and an indication of when each token begins and ends execution. As with *comview*, *planview* provides both "gestalt" views and interactive modes of operation. In fact, much of the user interface is shared by both tools (which also makes it easier for users to learn the new tools).

Each row in the *planview* display represents one timeline, and each rectangle is a token (Figure 6). The position and size of a token represents when it started and ended (or when it is *expected* to start and end, if that is in the future). The color of a token represents



Token Info

Token Name:	VAL-1586
Timeline Name:	OBS_WINDOW
Start Window:	(10.00 892.00)
Start Earliest:	10.00
End Window:	(910.00 1792.00)
End Earliest:	910.00
Duration Window:	(900.00 900.00)
Type:	((OBS_WINDOW_OP_NAV (9 00)))

Constraints for token VAL-1586

```
4490.00     *ENDS_BEFORE VAL-010
4490.00     *ENDS_BEFORE VAL-007
2690.00     *BEFORE VAL-2438
2690.00     *ENDS_BEFORE VAL-022
2690.00     *BEFORE VAL-2414

501.00       ENDS_AFTER_START VAL-1881
501.00       ENDS_AFTER VAL-1624
0.00         ENDS_AT_START_OF VAL-1722
```

Figure 7. Token and Constraint Descriptions

its status (active, completed, violated, etc.). When a token rectangle is selected, more detailed information about the token is displayed textually in separate windows (Figure 7). Selecting a temporal constraint in the text window (Figure 7) causes the constraint to be displayed in the graphical window (Figure 8).

As *planview* processes a log file, token rectangles change in color, size, and location. More importantly, the tool automatically propagates the temporal constraints through the token structure. For example, if the executive starts to achieve token **A** at time 150 and the expected duration of the token is the range [50, 100] seconds, then *planview* determines that the token must end in the range [150, 200]. Similarly, if token **B** is constrained to start after token **A** ends, then *planview* can determine that token **B** must start after time 150. By propagating these constraints as tokens begin and end, *planview* can detect faults, such as cases where a constraint is actually violated (e.g., if for some reason the executive were to begin token **B** at time 125), or cases where the start or end timepoint is projected to fall outside its respective window. Faults are flagged by changing the color of the affected tokens in the graphical display and by highlighting the affected constraints and/or time windows in the textual display.

Detecting a constraint violation is only the first step, however. The root cause of the problem must still be determined. For instance, one token ending late may have a ripple effect that eventually forces a subsequent token to begin late. While the interactive browsing facilities of *planview* enable users to follow

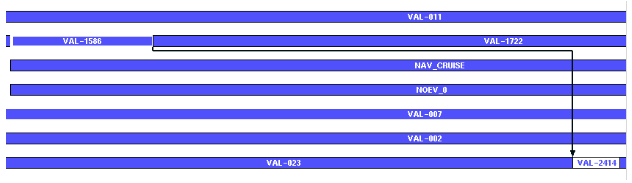
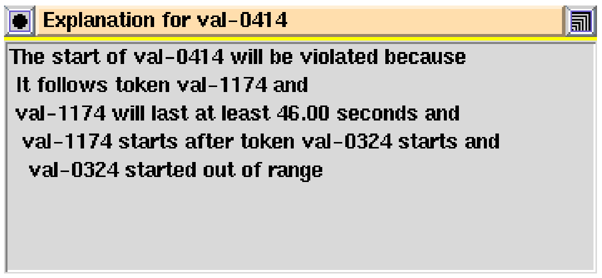


Figure 8. Visualizing Constraints Between Tokens



Explanation for val-0414

```
The start of val-0414 will be violated because
 It follows token val-1174 and
 val-1174 will last at least 46.00 seconds and
  val-1174 starts after token val-0324 starts and
   val-0324 started out of range
```

Figure 9. Automatically Generated Explanation

the chain of constraints manually, we have implemented a facility that reduces the need for this by automatically generating English-language explanations for why a given constraint is violated (Figure 9). By selecting a line in the explanation window, the appropriate constraint and tokens are highlighted in the graphical display. Briefly, *planview* generates an explanation by annotating which constraints were used for propagation, and then uses those annotations to form a tree of dependencies between tokens. The process is dynamic, so that when information changes and new constraint propagations occur, the explanation is automatically updated (for instance, if a different constraint causes the projected start time of a token to be even later than initially expected).

4. Related Work

The ParaGraph tool [4] provides a variety of visualizations of different aspects of a parallel system. ParaGraph animates trace information from actual runs to display behavior and provide graphical performance summaries. A task Gantt chart depicts the activity of processors with horizontal segments resembling the graphical display of *comview.* Together with the space-time diagram, showing interprocessor-communication, much of the functionality of *comview* can be duplicated. The primary differences between the tools are that *comview* integrates the two ParaGraph displays into a single display, and it provides extensive interactive features. These features allow the user to step through and closely examine individual communication and task events (and various aspects of those events) and correlate the events with their record in the log file.

The combination of the Mach Kernel Monitor and the PIE tool [6] demonstrates the visualization of parallel and distributed algorithms and their interaction with the operating system. Specifically, the tools enabled the authors to examine various kernel scheduling algorithms and observe the resulting performance of a matrix multiplication. The work clearly demonstrates the usefulness of a timeline-style visual tool for monitoring, debugging, and performance analysis of a concurrent system.

tnfview [5] is a tool designed for visualizing threads under Solaris 2 that are traced using logs written in Trace Normal Format. The display shows activities that include thread state transitions and context switching in a timeline-style format. It is intended to allow visual relation of concurrent thread activity such as examining the timing of thread switching as they interact via a condition variable. The tool is part of a suite of tools that can be used for debugging and performance analysis of multi-threaded programs.

A broader discussion of program visualization techniques for parallel and distributed programs is provided in [1].

5. Conclusions and Future Work

This paper has presented two visualization tools that aid in the validation of concurrent, distributed autonomous systems. The tools embody a mixed-initiative approach to problem solving: The tools automate what machines do best (massaging large amounts of data, presenting them in easily understood forms, and maintaining relationships between data) and facilitate what humans do best (spotting patterns and tracking down root causes of problems). The combination results in a powerful and flexible framework for software validation.

The *comview* tool, which is used to find timing, bottleneck, and interface problems, works with a fairly general message passing architecture, including both publish/subscribe (broadcast) and client/server (query) type messages. As such, it should be easy to use the tool with other message passing systems by instrumenting them and providing a log file parser. On the other hand, the *planview* tool, which is used to detect problems in plan execution, is fairly specific to the NASA *Remote Agent* executive. It presumes a plan representation based on timelines, tokens and temporal constraints. It is not clear how widespread this type of plan representation is, or how easy it would be to adapt *planview* to work with other types of plan representations (such as the hierarchical task decomposition used by the Task Control Architecture [10]).

Our current work is in making the tools even more useful, both by extending their range of interactive options and by incorporating analysis algorithms that can detect some classes of problems automatically. In particular, in conjunction with NASA, we are exploring methods for representing complex patterns and automatically detecting violations of those patterns in the log files. We are also interested in making the tools more suitable for real-time monitoring of system execution, which primarily involves speeding up the graphics (or finding more selective ways of displaying them).

Although these tools are just beginning to be used, they have already elicited a fair amount of support among developers. As currently fashioned, they take much of the drudge work out of analyzing the message and executive log files to find both simple and subtle bugs in the software system. Although visualization tools are not a panacea, we believe that good tool support will significantly aid developers in achieving the goal of "faster, better, cheaper," and more autonomous, spacecraft.

Acknowledgments

This research was sponsored by NASA, under contract 960322. We gratefully thank the members of the DS1 software team for their support and feedback.

References

[1] W. Appelbe, J. Stasko, E. Kraemer. Applying Program Visualization Techniques to Aid Parallel and Distributed Program Development. TR GIT-GVU-91-08, Graphics Visualization and Usability Center, Georgia Institute of Technology, July 1991.

[2] D. Bernard and B. Pell. Designed for Autonomy: Remote Agent for the New Millennium Program. In *Proc. i-SAIRAS '97,* Tokyo Japan (this volume).

[3] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp,D. Miller, and M. Slack. A Proven Three-tiered Architecture for Programming Autonomous Robots. *Journal of Experimental and Theoretical Artificial Intelligence*, **9:2**, 1997.

[4] M. Heath, J. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, **8(5):29-39**, Sep. 1991.

[5] S. Kleiman, D. Shah, B. Smaalders. Programming with Threads. SunSoft Press, Mountain View, Ca., 1996.

[6] T. Lehr, D. Black, Z. Segall, D. Vrsalovic. MKM: Mach Kernel Monitor Description, Examples and Measurements. TR CMU-CS-89-131, Computer Science Department, Carnegie Mellon University. Mar. 1989.

[7] T. Mason and D. Brown. *lex & yacc*. O'Reilly and Associates, Sebastopol, CA, 1990.

[8] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Reading, MA, 1994.

[9] B. Pell, E. Gat, R. Keesing, N. Muscettola, B. Smith. Plan Execution for Autonomous Spacecraft. in *1996 AAAI Fall Symposium on Plan Execution: Problems and Issues*, TR FS-96-01, AAAI Press.

[10]R. Simmons. Structured Control for Autonomous Robots. *IEEE Transactions on Robotics and Automation*, **10:1**, Feb. 1994.