Using Sequence Diagrams to Detect Communication Problems between Systems

Mikael Lindvall, Chris Ackermann
Fraunhofer Center for Experimental Software Engineering Maryland (FC-MD)
4321 Hartwick rd, suite 500, College Park, MD 20740
mlindvall@fc-md.umd.edu, cackermann@fc-md.umd.edu

William C. Stratton, Deane E. Sibol
Johns Hopkins University Applied Physics Laboratory Space Department Ground Applications Group (APL)
11100 Johns Hopkins Road, MS 4-118, Laurel MD 20723-8099
William.Stratton@jhuapl.edu, Deane.Sibol@jhuapl.edu

Arnab Ray, Lyly Yonkwa, Jan Kresser (FC-MD)
array@fc-md.umd.edu, lyonkwa@fc-md.umd.edu, jkresser@fc-md.umd.edu

Sally Godfrey
Code 583, Bldg 23, E215
Goddard Space Flight Center (GSFC), Greenbelt, MD 20771
Sara.H.Godfrey@nasa.gov

Jens Knodel
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
Jens.Knodel@iese.fraunhofer.de

Abstract—Many software systems are evolving complex system of systems (SoS) for which inter-system communication is both mission-critical and error-prone. Such communication problems ideally would be detected before deployment. In a NASA-supported Software Assurance Research Program (SARP) project, we are researching a new approach addressing such problems. In this paper, we show that problems in the communication between two systems can be detected by using sequence diagrams to model the planned communication and by comparing the planned sequence to the actual sequence. We identify different kinds of problems that can be addressed by modeling the planned sequence using different level of abstractions.

TABLE OF CONTENTS

1———————————————

[1] 1-4244-1488-1/08/$25.00 ©2008 IEEE.

[2] IEEEAC paper#1099, Version 3, Updated 2007:12:11

INTRODUCTION

Many software systems are actually evolving complex system of systems (SoS) for which inter-system communication is both mission-critical and error-prone. Software failures in the communication between the participating systems in a SoS, e.g. between Flight Software and the Ground System, can cripple system capabilities, cause loss of data, and even result in mission failure. Such problems ideally would be detected before deployment, but current state-of-the-art technologies do not easily support their detection calling for new research.

A preliminary analysis of APL's Common Ground System (an evolving SoS) identified 15 trouble reports related to problems with inter-system communication that had adverse mission impacts and for which there were no workarounds. This finding motivates our research to develop automated support for architects and (I)V&V to check system communication across a SoS.

Previous efforts have analyzed static architectures [1] and have led to the development of a general understanding of dynamic architectures of small software systems [2], but they do not typically address the problems APL has been facing. In a NASA-supported Software Assurance Research Program (SARP) project called Architecture Analysis of Evolving Complex Systems of Systems, we are researching a new approach that will explicitly address such problems.

In this paper, we describe the background to this SARP project, the problems we are addressing and problems we encountered in the existing technology that had to be dealt with, aspects of the proposed solution, as well as the results from a study we conducted in order to determine the feasibility of the proposed solution.

BACKGROUND

In 2006, APL and FC-MD conducted a joint research infusion project. The infused technology, the Fraunhofer SAVE (Software Architecture Visualization and Evaluation) tool was applied to APL's Common Ground System (CGS). Fraunhofer SAVE is a joint development effort between FC-MD and Fraunhofer Institute for Experimental Software Engineering (IESE). The goal of the infusion project was to eliminate maintenance and evolution problems. All of APL's NASA missions use the CGS for spacecraft I&T and operations. CGS is currently supporting operations for three deep space missions: MESSENGER, STEREO, and New Horizons. Flight software, scientific data processing software, and ground equipment software interface with CGS and depend on its services. The infusion project provided results that are used to improve CGS's architecture [3]. As part of the technology infusion project, we identified a set of problems related to dynamic aspects of CGS that are beyond the scope of the basic SAVE technology, which is based on static analysis of source code.

TROUBLE REPORTS

For this SARP project, we conducted an analysis of APL's databases of Change Requests and Anomaly Reports in order to understand how common and serious these problems are in reality. The analysis focused on the Ground System and the 15 trouble reports, which are related to problems with system interfaces and system communication. Currently, there is a lack of research and technology to address these problems. In particular, there is no automated support to describe and compare expected dynamic profiles of a SoS with the actual dynamic profiles in order to detect problems. Dynamic profiles are characteristics of a system during runtime as observed through an interface, such as call and event ordering, data and data formats, resource consumption, and timing information.

While interfaces permit systems to communicate with each other, they are also often the source of problems. One reason for such problems is that systems are developed by different teams with different interpretations of interface specifications. Individual developers may occasionally attempt to include run-time checks in their code ensuring that interface specifications are followed, but there is often no systematic way for architects, or a V&V or IV&V team, to describe and check these rules in a consistent manner across a SoS. This is especially true in situations where systems are composed of third-party components, since it is often impossible to program such checks when the system has been developed and provided to the system integrator. These problems are typical for complex and evolving SoS.

THE PROPOSED SOLUTION

The proposed solution will address problems related to interfaces and the dynamic profiles of systems that communicate in runtime. The technology will allow the user to define and navigate the expected (a.k.a. planned, specified, desirable, ideal, baselined etc.) dynamic profile of a system as well as comparing it to the planned profile so that s/he may evaluate whether the system conforms to specifications or not. We are researching different ways to define and navigate these dynamic profiles. For example, we are researching adding dynamic information to static descriptions and diagrams and formulating ways to explicitly define interface rules. These interface descriptions will be based on the already existing interface specifications. In these specifications, e.g. the order of calls and events, the size and format of data, as well as timing information are specified. We are also investigating ways to describe and collect data about resource allocation, since resource allocation can be viewed as a communication between a system and the operating system. We consider using a combination of static connection diagrams, sequence diagrams, rules, and assertions to describe the dynamic profiles. We are researching ways to collect data from the system during run-time as well as ways to compare the expected dynamic profile with run-time data in order to detect problems.

The tool will be developed as an extension to the SAVE tool. SAVE is a tool that visualizes and compares the implemented software architecture (actual) with its planned architecture based on static analysis. One of the missing features of the SAVE tool in the context of SoS is its ability to detect dynamic linkages between systems. While analysis of individual systems is useful, there is also a need to analyze their interaction.

What makes such an analysis complex is that the communication between these subsystems is established only at run-time for which mere static analysis, as is done in SAVE, is inadequate. The problem is that static analysis, though sound, would report many false positives (spurious connections that do not actually exist), because static analysis always constructs conservative over-approximations. For example, if two components listen to ports defined from the same sets of ports, then static analysis would flag a dependency even if the ports they listened to did not have the same port id. One important part of this project is to research ways to detect such dynamic dependencies.

Because SAVE has some of the needed capabilities (e.g. features for comparison of expected and actual architectures, visualization, navigation, etc), we will reuse its basic functionality, add a dynamic dependency detection and visualization component, and use a combination of static and dynamic analysis to extract the actual architecture from the code. At a very high level, this consists of taking a set of executions of the system then deducing dynamic dependencies based on trace analysis. In addition, SAVE's approach for defining, visualizing, and navigating planned static architectures and comparing them with the actual has been shown to be very beneficial in other projects and thus provides a good foundation for this research. Thus, SAVE already has basic support for some of the required features that need to be added. However, before SAVE can detect problems in the interfaces, the tool has to be improved because such dynamic couplings remain undetected and applications cannot be analyzed together.

THE COMMON GROUND SYSTEM (CGS)

CGS [4] consists of 83 different evolving systems (applications). These applications are developed, compiled, and launched independently from each other, and participate in a complex pipe-and-filter architecture established during run-time, see Figure 1.

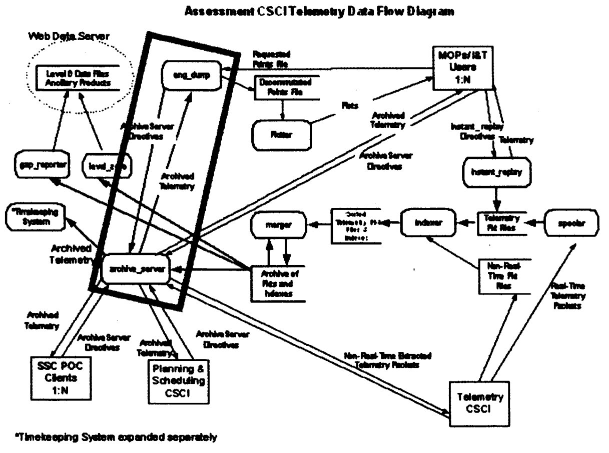


Figure 1 – ArchiveServer and EngDump in Common Ground's pipe&-filter architecture

In this paper, we focus our examination on two applications that are representative of the Common Ground software architecture and which belong to the Assessment sub-system, see rectangle in Figure 1:

- Archive Server – an application that serves selected telemetry packets from the archive
- Engineering Dump – a client to the archive server that extracts selected telemetry points from the archived packets, converts the raw telemetry points into engineering units using calibration data, and formats and stores the converted points in a file.
- Common – The two applications reuse source code through a component called common.

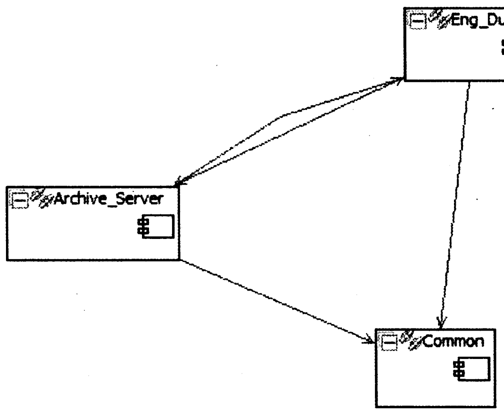


Figure 2 – Planned Architecture: Engineering Dump is a run-time client of Archive Server; both applications are built using shared modules stored in "Common."

The SAVE tool was used to analyze the structure of Archive Server and Engineering Dump. The planned architecture of the selected applications was modeled as shown in Figure 2. The figure illustrates that the Archive_Server and the Eng_Dump are dependent on each other and that they both use code from Common.

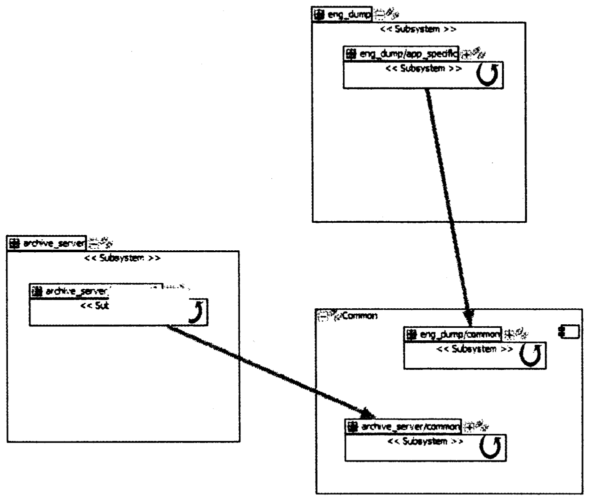


Figure 3 - Actual Architecture

The actual architecture of the selected applications in Figure 3 illustrates how the source code is organized in the file system. Each application has a folder called app_specific in which the code that is unique for that application is stored. Each application also has a folder called common. As the figure illustrates, there is one common folder for each application, but any file that exists in both folders are identical.

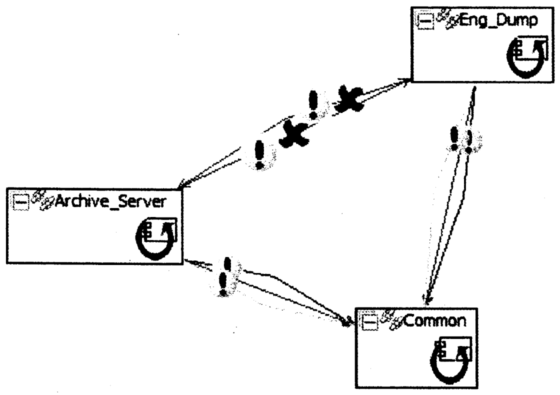


Figure 4 - The planned dynamic dependencies between Archive Server and Engineering Dump are not visible to the SAVE tool, but there are some spurious couplings.

The comparison of planned vs. actual architecture is shown in Figure 4, which is a combined diagram that illustrates several facts. On the one hand, the diagram illustrates the fact that the SAVE tool did not discover the direct dependencies between Engineering Dump and Archive Server. These "missing" couplings are illustrated by red crosses. SAVE does not detect these couplings because there are no static dependencies between the two applications since the two processes communicate dynamically via sockets and SAVE only detects dependencies that are present in the static code.

On the other hand, the diagram illustrates that there are spurious couplings between Engineering Dump and Archive Server. To examine coupling of the Archive Server and Engineering Dump through the common modules, we analyzed the code together with the SAVE tool. This caused some confusion in the tool because the applications are not supposed to be compiled together causing namespace conflicts to occur. For example, in several cases, the same name was used for a global variable that exists in both applications. Since SAVE assumes that all names are unique, spurious couplings were introduced between the two applications. The spurious couplings are illustrated by exclamation marks in the diagram.

In addition, there are some extra couplings between Engineering Dump and common as well as between Archive Server and common. These couplings are due to the fact that unexpected access and import dependencies are present in the source code.

Thus, before the SAVE tool can be used for analysis of more dynamic problems related to two applications that are based on dynamic dependencies, a strategy has to be developed that allows for such analysis, removes spurious couplings, and identifies missing couplings as the ones described above. The strategy we discuss in this paper avoids these issues by viewing each application as a black box and by only focusing on the communication that occurs between the client and the server. In the future, we plan to develop a strategy that allows an analyst to move freely between the applications of such a system and analyze both structure and behavior on arbitrary levels of abstraction.

PROBLEMS FROM TROUBLE REPORTS

The analysis of APL's databases of Change Requests and Anomaly Reports identified 15 problems that could potentially be detected using dynamic analysis. These problems were discussed in a series of workshops with APL. Following are examples that are representative of such problems.

APL has experienced several significant problems in getting the CGS to work with other systems such as the Flight Software, the Jet Propulsion System (JPL), Deep Space Mission System (DSMS), the APL Ground Support Equipment (GSE), and the remote Payload Operations Centers (POCs). Interface Control Documents (ICD) govern the communications with each of these external systems. However, APL frequently encounters systems that do not comply with ICD's. Another example is the Archive Server and the Engineering Dump. The Archive Server serves many other clients beyond the Engineering Dump. Other teams often develop those other clients and since there is little or no communication between the different teams, interface problems are introduced that are difficult to detect. APL conducts integration testing to a large extent, but problems still occur in operations because of incorrect use of communication protocols.

A related example is a failure that occurred during the communication between the ground system and a satellite using the file transfer system CFDP. The satellite system was supposed to send a series of files and the ground system was supposed to acknowledge receipt of those files. A software valve had been added to the system so that the operators would be able to block uplink of CFDP acknowledgement messages while more important commands were being sent since bandwidth is always limited. At one point, the valve was closed and when the ground system started to recognize that file data was missing, it responded with NACKs (not acknowledged). However, due to the closed valve, these NACKs could not be transmitted and were stuck in a queue. The NACKs could only go through later when the valve was opened. Once the valve was opened, the NACKs caused the satellite system to send all the missing messages again. In fact, there were multiple NACKs for each file in the queue since a NACK is resent after each communication timeout. Thus, the missing file data was sent multiple times, crowding the connection and misusing the valuable bandwidth.

It should be noted that in this example, both systems behaved according to the specified protocol. In addition, this problem was only detected by chance during an unrelated investigation, illustrating how difficult it is to detect such problems. What is missing is a way to analyze the behavior from the viewpoint of the architecture. From this viewpoint, a high number of NACK's is an indication that there is misbehavior. Another similar example that was mentioned is that in one case, the client connected to and disconnected from the server 4,000 times during a short time. On a micro level, this behavior matches the specification, but on a macro level it does not. The desired macro level behavior is to connect, stay connected, and only disconnect when there is no need to stay connected anymore.

In both these examples, the main concern is intersystem communication, i.e. the communication of APL software with other systems, such as the satellite software. APL's interest is in verifying whether systems follow the communication procedures according to the ICD. However, mainly only micro level behavior (short sequences) is specified in the ICD while macro level or architectural behavior is not. Issues that frequently occur are ones related to

- Message sequencing
- Content of control messages
- Timing of messages

Message sequencing is related to the order in which the messages are supposed to occur. Problems occur when messages appear in a different order than the specified one. Problems occur when the system expects a certain message but receives a different one. Timing of messages can be important, especially in situations when a message has to arrive within a certain time period. Problems also occur when messages do not arrive within the specified time period. Especially data messages can be large in size, but most of the time only the header of such a message is interesting for analysis, not the data itself. In this SARP project, we have started addressing problems related to message sequencing and content of control messages. In the future, we will address timing problems.

THE STUDY

Dynamic Compliance Checking Approach

We studied whether our proposed solution would be feasible for detecting problems stemming from deviations from interface control documents. More specifically, we studied whether it would be possible to compare a planned sequence diagram with actual sequences in order to detect deviations. We were especially interested in identifying different kinds of communication problems, for example in terms of how much modeling would be necessary to detect them. Software development teams often lacks time and resources and if such modeling requires too much effort and too much modeling background, chances are that a new technology that relies on such modeling will never be used.

The study was conducted in the following way.

(1) The APL team produced a sequence diagram that specifies the planned communication between the server and the client. This sequence diagram was based on information provided in the ICD.

(2) The APL team then captured dynamic data from a correct communication between server and the client. The communication was correct in that sense that it matches the planned sequence diagram (with one exception, see below).

(3) The FC-MD team developed a parser based on the ICD and dynamic data provided. The parser reads the dynamic data and outputs the messages that were sent between the two applications. For each message, the timestamp, the message type, and the message content were provided.

(4) Once it was determined that the parser worked correctly, the APL team produced a set of three communication sequences that each was not compliant with the planned sequence in one of the following ways: 1. There were missing messages, 2) there were extra messages, or 3) there were messages whose parameter values were inconsistent with the specification. The defects were specified by the APL team as well as the actual and correct system behavior.

(5) The FC-MD team imported the sequences into SAVE and used the new SAVE prototype extension in combination with manual analysis to compare the planned sequence diagram to each of the actual sequence diagrams.

(6) Deviations between the planned and the actual sequences were analyzed, documented, and reported to the APL team.

(7) The APL team determined whether the detected deviations were true or false.

(8) The APL team and the FC-MD discussed the feasibility of the proposed approach and potential improvements to make it useable in a "live" situation at APL.

The next subsections will discuss two of the three sequence diagrams. First, the specification will be presented, and then the evaluation and analysis results are discussed.

Specification: The protocol as a high level planned sequence diagram

The client (eng_dump) and the server (archive_server) communicate using a protocol that specifies four different types of messages:

(1) The client defines a set of filters that together specifies the type of data that it requests the server to return. Examples of filters are: Type of data, e.g. STP or TP, and Time range, specified by start time and stop time. The filters from the client to the server can be sent in arbitrary order.

(2) When the client has specified, using filters, what data to download, it sends a BeginPlayBack command to the server. Once this command has been issued, the client is not expected to send more messages.

(3) As soon as the BeginPlayBack message has been received, the server starts sending data messages. Each data message must match the filter specification received earlier. For example, the type of each data message must be as specified and the time stamp of each message must fall within the specified time range.

(4) When there are no more messages, the server sends an End Of Transmission (EOT) message to the client signaling that the data transmission is complete; thereafter the communication link is closed.

Since the goal is to develop a modeling and evaluation method that requires limited effort, we started by modeling the protocol using high level sequence diagrams, see Figure 5. This model models the facts that there might be several filters followed by one BeginPlayback message, and that there might be several data messages followed by one EOT message. This high level model (since it does not provide any information about the type of filters or data messages that we expect to occur, is called "high level planned sequence diagram.")

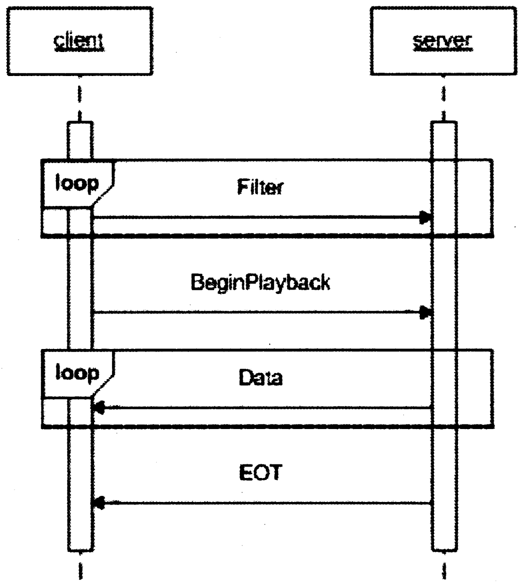


Figure 5 - High-level planned sequence diagram

Evaluation based on the high-level sequence diagram

The next step was to evaluate sequences based on the planned sequence diagram. We started by applying it to the first sequence that we received. This sequence is expected to be correct, i.e. we expected it to match the specification without any extra or missing messages. The evaluation result is provided in Figure 7. As one can see in that figure, the final message EOT appears to be missing from the actual sequence. This is surprising because this sequence was supposed to be correct. The APL team analyzed the original 'nominal case' data and even re-ran the test example to verify that everything was correct. Still, it was difficult to understand why the EOT is not being sent from the archive server nor how the client knows when to close the socket. This can be a significant problem because clients are expected to close the socket to the server once they have received all the requested data, as indicated by the EOT message. If the EOT message is not being used for its intended purpose, clients may be employing some other means of recognizing that all the resulting data has been received. For example, they might check that the time stamp on the data is equal to exceeds the stop time specified. For a variety of reasons, including timestamp precision, multiple of packets from different paths and/or sources with the same timestamp, etc., this may lead to the client prematurely closing the socket and missing data subsequently to be returned by the server. During extended analysis, the APL team realized that the missing EOT message is actually not missing at all. Because the TCP packets, as reported by snoop, can aggregate data, the EOT message is actually at the end of the last TCP packet from the server to the client (i.e., the server produced two write statements for the STP and EOT, but it ended up as one

TCP packet). This is perfectly legal, with Nagles algorithm enabled. Our issue in the analysis was that we were treating each snoop TCP packet as a message. This illustrates that our parsing algorithm has to be able to detect aggregated data situations.

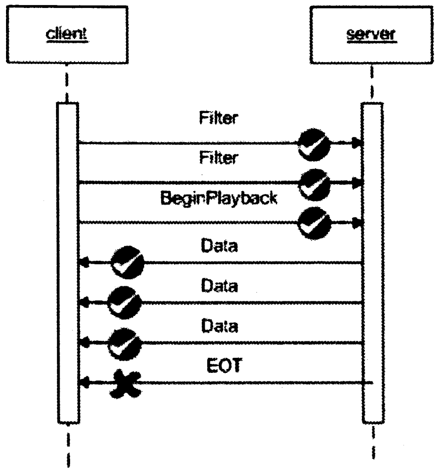


Figure 6 - EOT is undetected

In the next example, it was detected that a filter change/addition was sent from the client to the server after BeginPlayback was sent while data from server to client was flowing. This filter message is ignored by the server (in fact, it might not even get read off the socket). The server assumes that clients read as fast they can, which could be severe problem if that is not the case. Thus, the risk is that the server gets stuck because that one client blocks the server. In addition, the client might expect the ignored filter to be in effect and thus receives the wrong data.

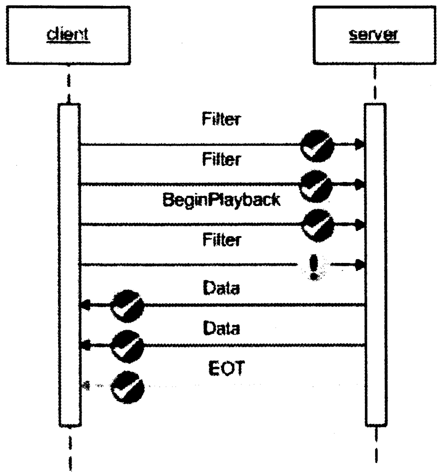


Figure 7. An illegal extra filter is sent after BeginPlayback message has been sent.

Specification: The protocol as a low-level planned sequence diagram

We proceeded by adding more information to the sequence diagram, see Figure 8. Two rules were modeled. 1. The rule that specifies that start time must be less that stop time, as well as 2. The rule that the data type of each of the received data messages must be the same as the specified type. We modeled these rules as assertions and added parameters to the messages. The parameters are used by the assertions and the assertions are evaluated for each message. Since we added more information to the model, we also needed to specify that the order between the different filters is not important. This is denoted by adding a star "*" in front of each filter. In addition, we needed to express the fact that there might be filters of other kinds than the ones we focus on. We express this by adding a general filter with stars as parameters. Since we needed to connect the data messages to the filter messages, we added a parameter Type.

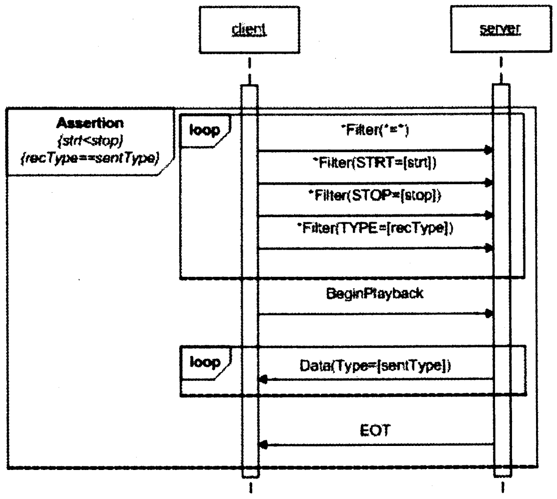


Figure 8 - Low level planned sequence diagram

Evaluation based on the low-level sequence diagram

We evaluated the captured sequences by applying the low level planned sequence diagram and matching it to the actual sequence. We use a lightning symbol to indicate that the names of the messages are correct but that there is a mismatch between the parameters of the messages. Thus the lightning symbol in Figure 9 indicates that the time specified in the 'STRT' (start time) was after the 'STOP' (stop time). Unfortunately, the ICD does not allow for an indication from the server to the client that it has specified an invalid filter. So a close() from the server, without an EOT message, is actually expected. Thus, though not a nominal case, this case illustrates compliance to the protocol defined in the ICD.

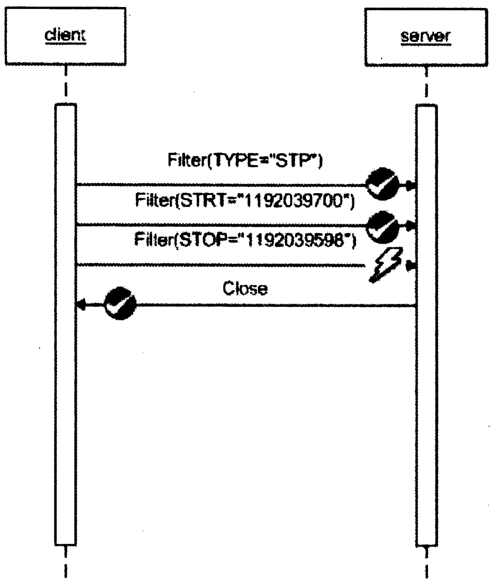


Figure 9 - Start time occurs after Stop time.

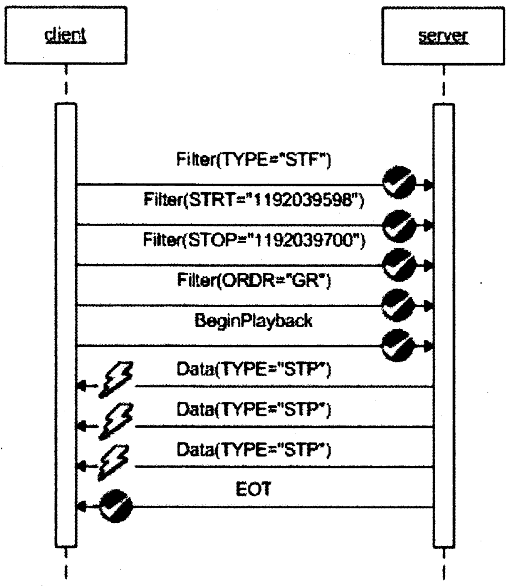


Figure 10. "STF" was requested "STP" was received

In the last example, it was detected that data messages of the wrong type were sent to the client. The root cause is actually that a bad filter is sent from the client to the server: TYPE:STF. STF used to be a supported type, but no longer is. Old legacy clients might still request STF-data, if they have not updated to the new ICD. Since STF is not a valid option, the server returns STPs, which is not expected. The server simply ignored the invalid filter and used its default which is STP. The correct server behavior should have been to close the socket on an invalid filter. Dependent on how robustly the client was implemented, this can be a significant problem because the client might assume that same type of data that was requested will be returned and process the incoming data according to that type. In fact, several internal clients, including Eng_Dump, are indeed coded in this fashion and do not verify that the type returned and about to be processed is of the type requested. Obviously, processing data of the incorrect type will cause incorrect data to be generated or even client application crashes.

CONCLUSIONS

The SARP project develops an approach for dynamic compliance checking and visualizing the results using sequence diagrams. The approach has been implemented as an extension to the SAVE tool and was validated in a first pilot study for APL's CGS.

The results from the study show that problems in the communication between two systems can be detected by using sequence diagrams to model the planned or expected communication and by comparing the planned sequence to the actual sequence. The results also show that there are different kinds of problems and that they can be addressed by modeling the planned sequence using different level of details. Sequencing problems, that is, messages that occur unexpectedly or out of order, can be detected by using high level sequence diagrams without details. Content problems, that is, problems which are related to the content of messages rather than to the order of messages require a more detailed modeling approach. The suggested approach, which is based on assertions in combination with sequence diagrams, seems to be a feasible approach for this problem. High-level modeling may be used without low-level modeling and vice-versa, allowing the user full flexibility over of the amount of time and resources he/she chooses to use the tool to detect such issues. The fact that only simple, standard modeling skills are necessary to become immediately productive with the proposed tool makes the approach appealing.

References

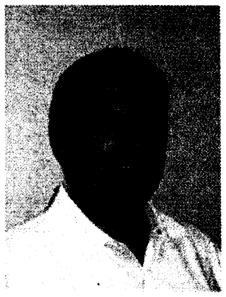
[1] Knodel, Jens, Lindvall, Mikael, Muthig, Dirk, and Naab, Matthias Static Evaluation of Software Architectures 2006, 279-294. Conference on Software Maintenance and Reengineering, CSMR 2006.

[2] Claudio Riva, Jordi Vidal Rodriguez, "Combining Static and Dynamic Views for Architecture Reconstruction" Conference on Software Maintenance and Reengineering , 2002 and Hong Y. et al. "DiscoTect: a system for discovering architectures from running systems," ICSE 2004.

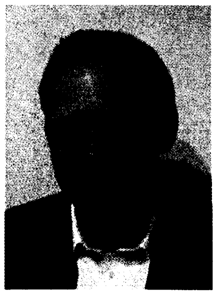
[3] William C. Stratton, Deane E. Sibol, Mikael Lindvall, and Patricia Costa, Technology Infusion of the SAVE Tool into the Common Ground Software Development Process for NASA Missions at JHU/APL, IEEE Aerospace Conference, 2007.

[4] McKerracher, P, Tillman, D, Furrow R M, Herrera, L., Complete Ground Software Re-use: The Common Ground Approach to a Re-usable, Shared Ground System. The Fifth International Symposium on Reducing the Cost of Spacecraft Ground Systems and Operations (RCSGSO). July 8-11, 2003.

Biographies

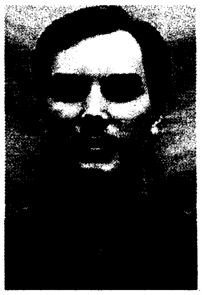


Dr. Mikael Lindvall is a senior scientist and the director of the software architecture and embedded systems division at Fraunhofer Center for Experimental Software Engineering Maryland. He is interested in best practices and methodologies for software engineering in general and specializes on software architecture evaluation and software evolution. He received his PhD in computer science from Linköpings University, Sweden in 1997. Lindvall's PhD work focused on evolution of object-oriented systems and was based on a commercial development project at Ericsson Radio in Sweden.

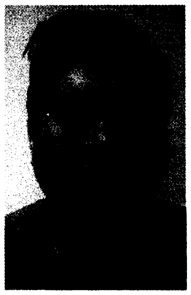


Mr. Christopher Ackermann is a scientist at the Fraunhofer Center for Experimental Software Engineering, Maryland and is pursuing a Ph.D. in Computer Science at the University of Maryland, College Park. He received his Bachelor's Degree from the University for Applied Sciences Mannheim, Germany in 2006. He has been active in the fields of software design, change impact analysis, and software verification. His current research interests include empirical software engineering, static and dynamic software analysis and verification.

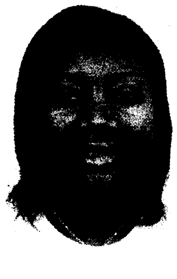
Mr. Bill Stratton is a Senior Professional Staff member in the Ground Applications Group of the JHU/APL Space Department, currently serving as the Radiation Belt Storm Probes (RBSP) Ground Software Lead Engineer. For the past six years, he has developed Common Ground software and contributed to the ground software portion of the Space Department's Software Development Process. Bill holds a BsC in Applied Mathematics from Brown University, 1977, and a Masters in Computer Science with a concentration in Software Engineering from JHU, 2003.



Mr. Deane Sibol is a Senior Professional Staff member in the Ground Applications Group of the JHU/APL Space Department, currently serving as the Radiation Belt Storm Probes (RBSP) Ground Software Deputy Lead Engineer. Prior to this role, he served as the Common Ground Software Lead Engineer, overseeing the overall ground software architecture and directing the ground software development for four (4) NASA missions. He holds a BS in Computer Science from Loyola College, 1993, and a MS in Computer Science from Old Dominion University, 1995.



Dr. Arnab Ray is a research scientist at Fraunhofer Center for Experimental Software Engineering Maryland. He is interested in formal approaches in software architecture, requirements engineering, design security and has experience in formal modeling/verification of automotive and medical device software. He holds a Bachelor of Computer Science and Engineering from Jadavpur University, India (1999) and a PhD from State University of New York at Stonybrook (2004) where his dissertation focused on compositional semantics of input notations used for modeling distributed systems.



Ms. Lyly Yonkwa has been working on design and development of software applications as well as in research on various software engineering areas since 2005. She is currently a scientist at the Fraunhofer Center for Experimental

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Using Sequence Diagrams to Detect Communication Problems between Systems - a feasibility study

IEEE Aerospace Conference 2008

Mikael Lindvall (FC-MD)
Chris Ackermann (FC-MD)
William C. Stratton (APL)
Deane E. Sibol (APL)
Arnab Ray (FC-MD)
Lyly Yonkwa (FC-MD)
Sally Godfrey (GSFC)

Fraunhofer Center for Experimental Software Engineering Maryland (FC-MD)
Johns Hopkins University Applied Physics Laboratory Space Department Ground Applications Group (APL)
Goddard Space Flight Center (GSFC)

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Project Goal

- Goal
  - To research and develop a tool for *run-time* architecture analysis
- The new tool, Dyn-SAVE,
  - will extend the already existing *static* Software Architecture Visualization and Evaluation (SAVE) tool
- Background
  - SAVE successfully applied to JHU/APL's Common Ground System in 2006 NASA Research Infusion project
  - Architecture = structure + behavior
  - Need for dynamic architecture analysis was identified
  - NASA IV&V support for a Software Assurance Research Project (SARP) to develop such a tool

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Motivation

- Systems are often difficult to understand, e.g.
  - Static and dynamic architecture very different
  - Distributed systems of systems hard to understand
- System verification is difficult, e.g.
  - Interface Control Documents interpreted differently
  - Changes of COTS behavior make upgrading risky

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

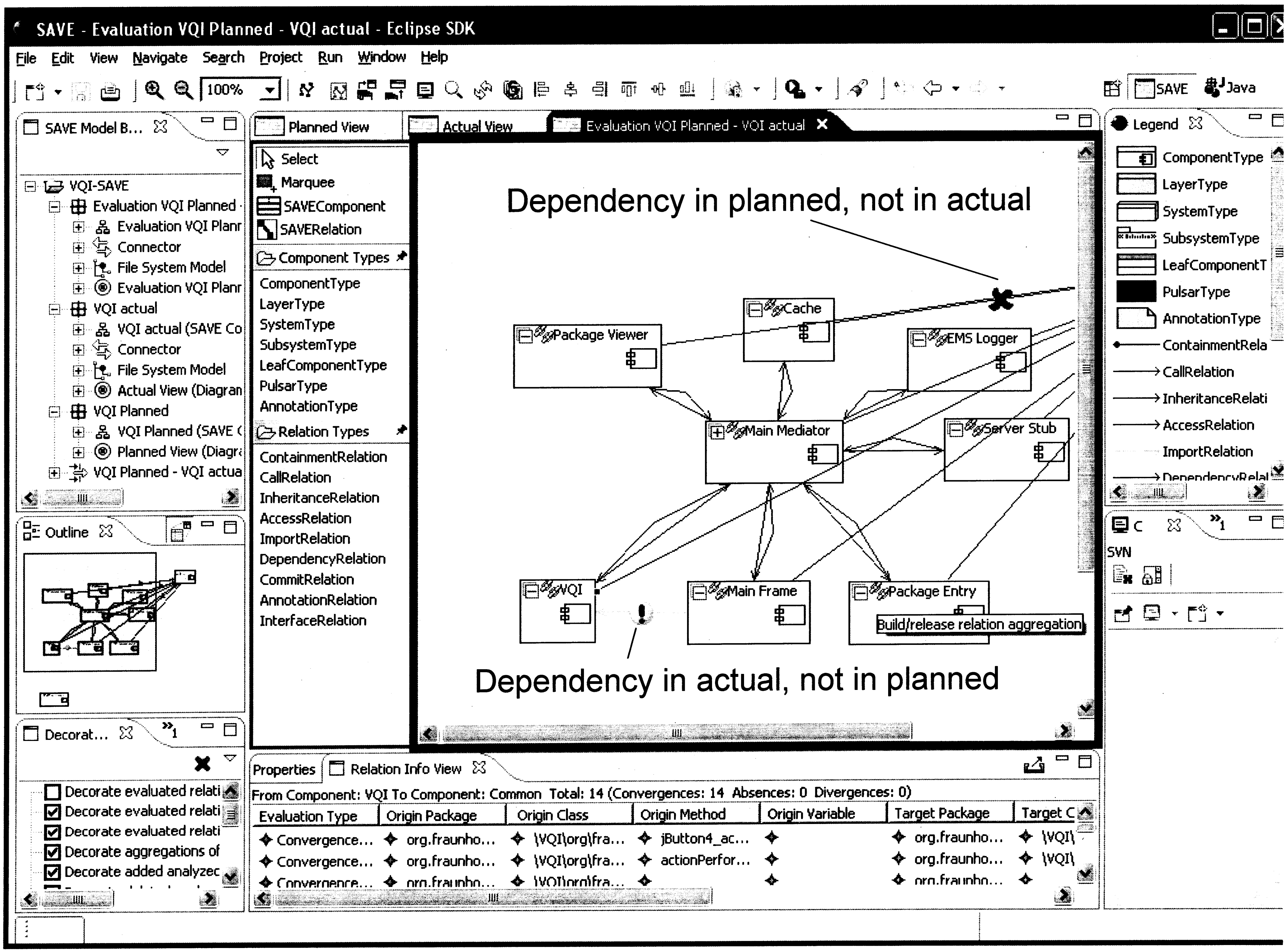
The (static) SAVE Tool

- Objective: Make Architecture/Design specifications alive!

- Helps answer: Does the implementation match the plan?
  - Define a *planned* (and/or target) architecture (using rules etc);
  - Create an *actual* architecture from source code;
  - Compare planned architecture w/ actual, identifying architectural violations

- Features for Zooming, Filtering, Refactoring
- Language independent: C/C++, Java, Delphi, Ada, Simulink, Fortran

- Conclusion after applying SAVE at APL and to many other systems:
  - The SAVE approach is useful and practical
  - One can quickly model and analyze software architectures
  - But has some weaknesses since it's based on static analysis

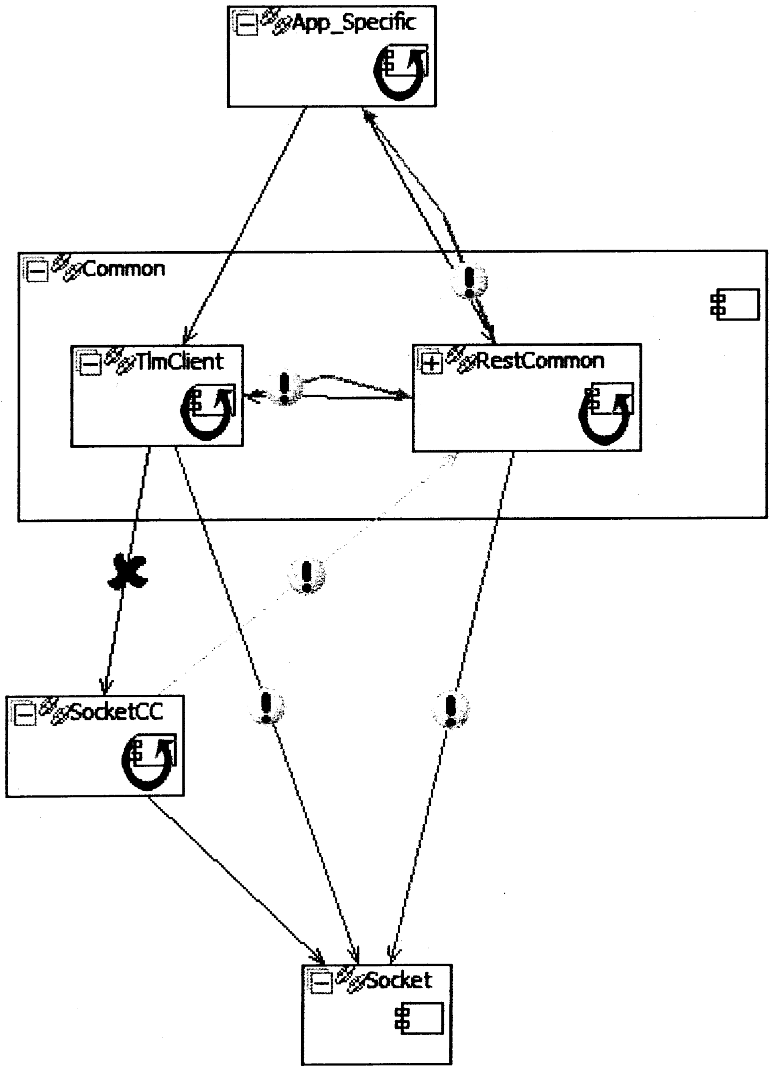


From Component: VQI To Component: Common Total: 14 (Convergences: 14 Absences: 0 Divergences: 0)

Evaluation Type	Origin Package	Origin Class	Origin Method	Origin Variable	Target Package	Target C
✦ Convergence...	✦ org.fraunho...	✦ \VQI\org\fra...	✦ jButton4_ac...	✦	✦ org.fraunho...	✦ \VQI\
✦ Convergence...	✦ org.fraunho...	✦ \VQI\org\fra...	✦ actionPerfor...	✦	✦ org.fraunho...	✦ \VQI\
✦ Convergence...	✦ org.fraunho...	✦ \VQI\org\fra...	✦	✦	✦ org.fraunho...	✦

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Current (static) SAVE Capabilities



Using static SAVE, we can identify some violations, but

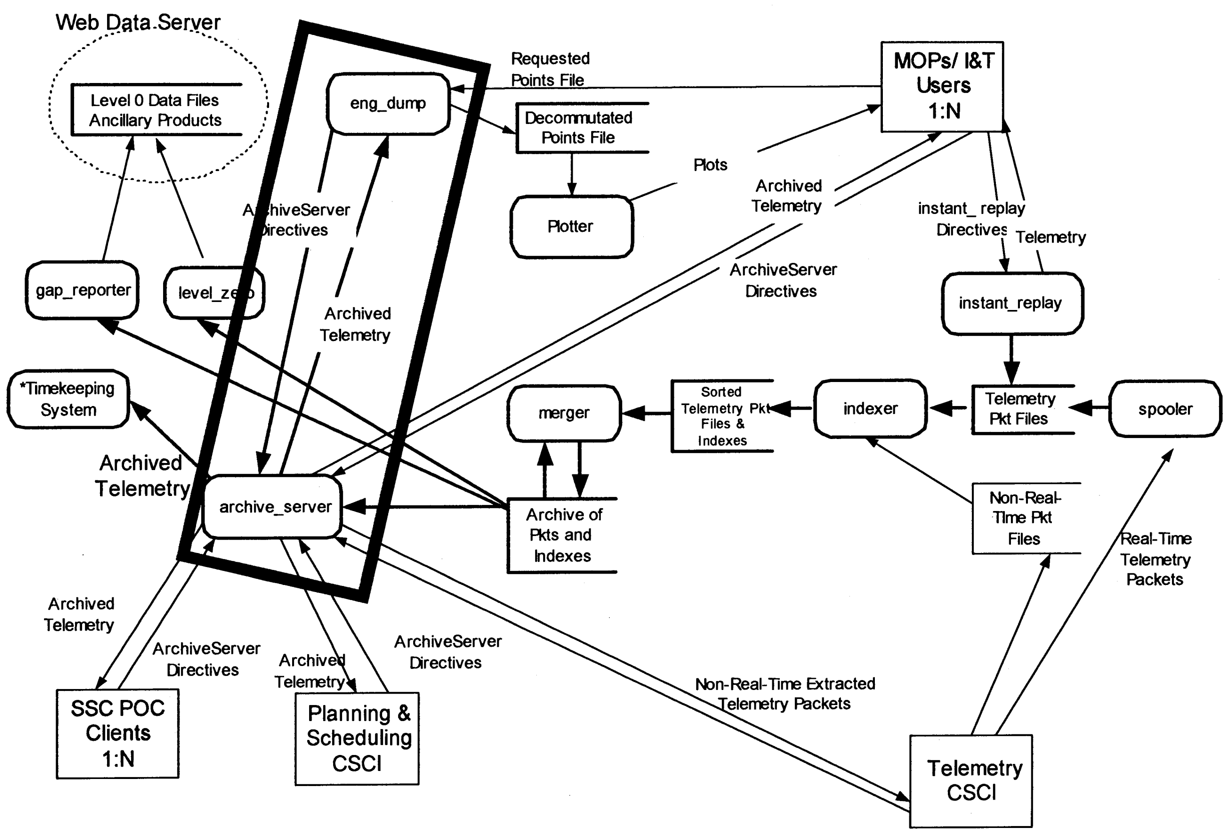
1. What are the implications of these couplings?
2. In what order do the couplings occur?
3. Who does socket communicate with?
4. Is communication according to specification?
5. What components are responsible for communication?

Let's see how these issues **could** be analyzed in the **future** using Dyn-SAVE!

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

The Common Ground System

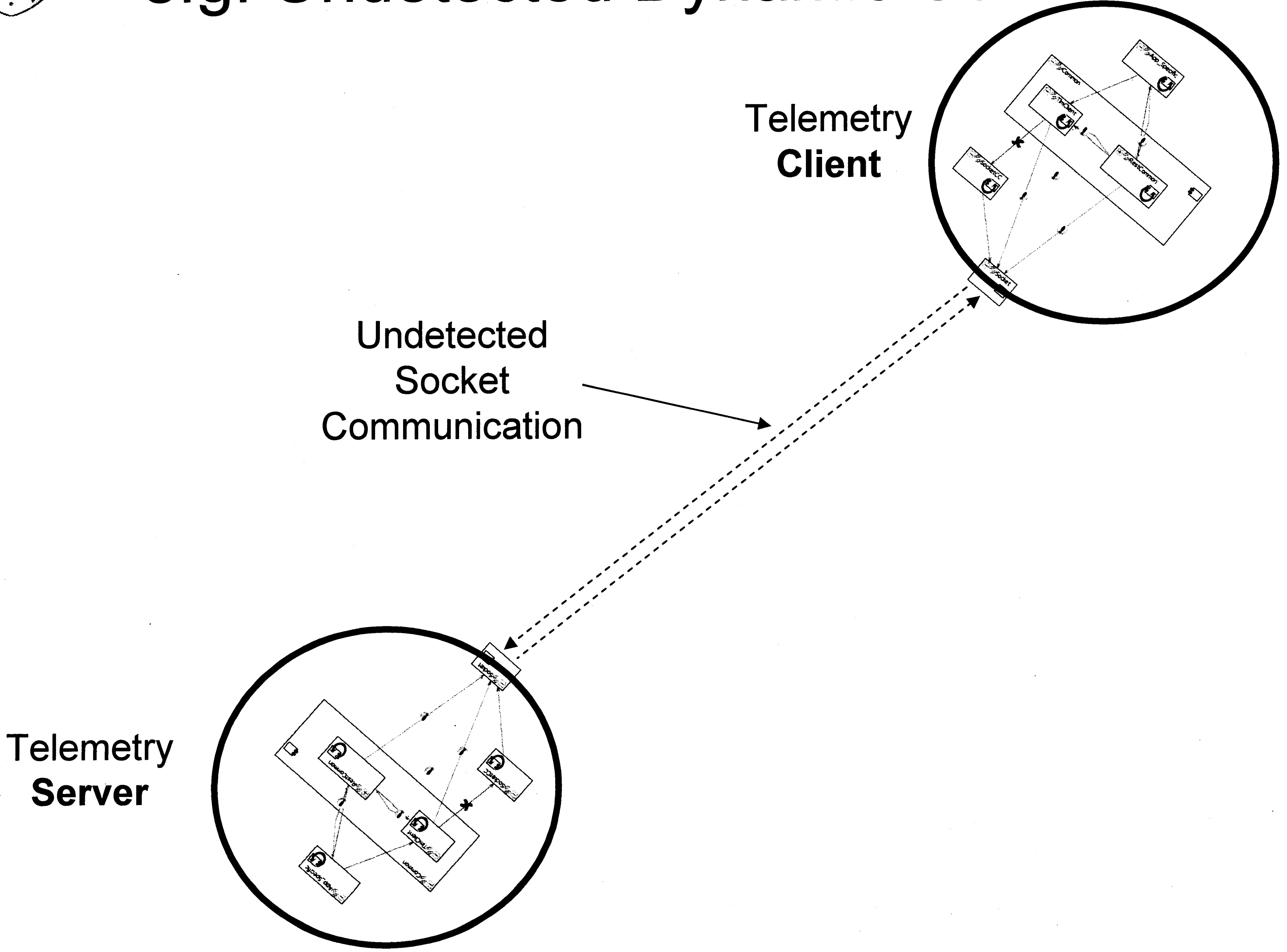
Assessment CSCI Telemetry Data Flow Diagram



*Timekeeping System expanded separately

LHerrera 08/03

Static Drawbacks e.g. Undetected Dynamic Connections



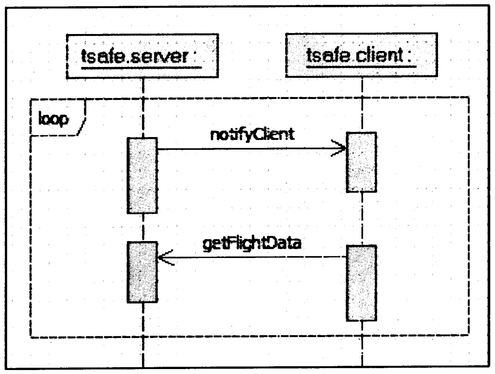
Dyn-SAVE Capabilities (Vision)

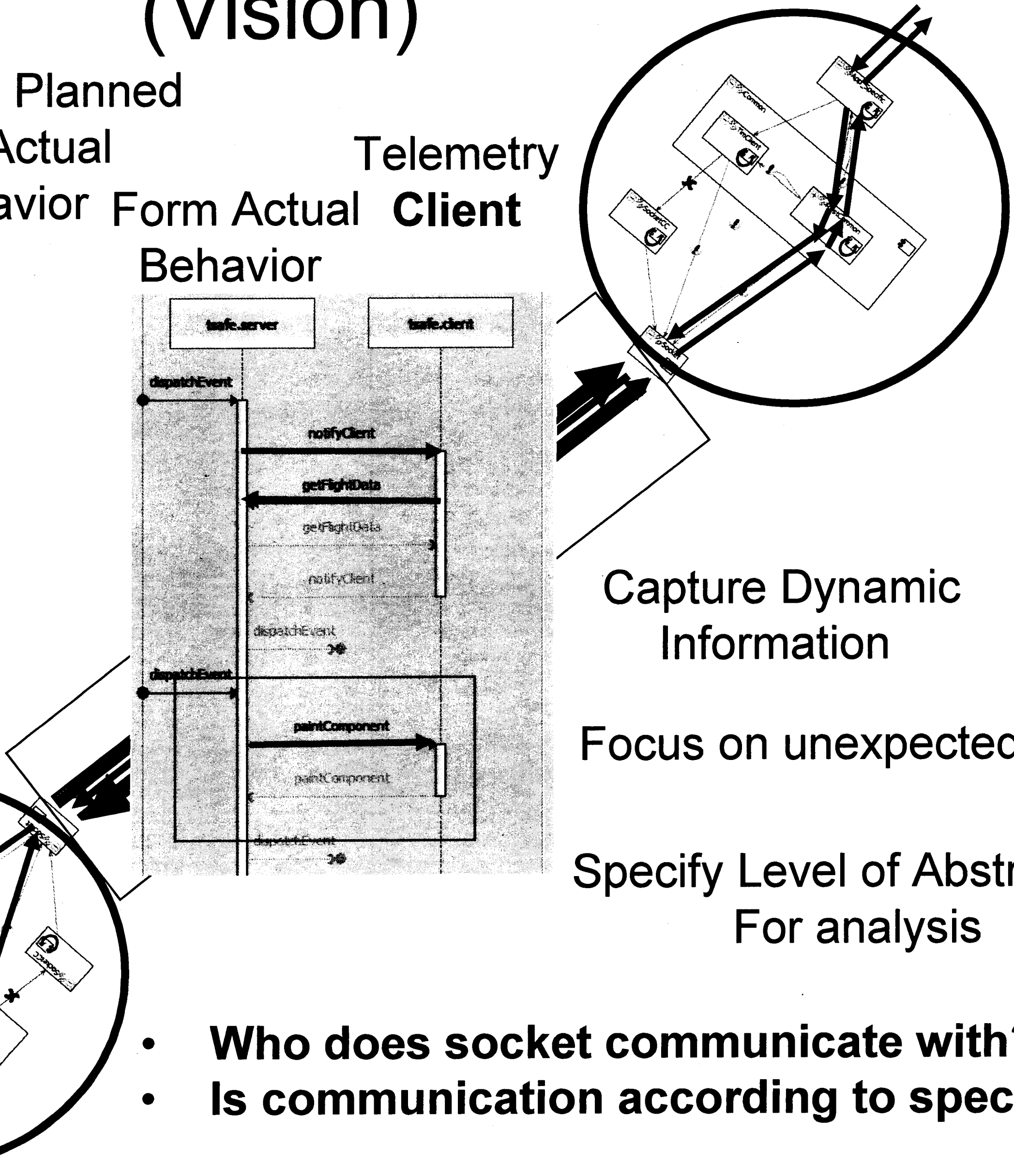
Compare Planned and Actual Behavior

Telemetry **Client**

Form Actual Behavior

Specify Planned Behavior





Capture Dynamic Information

Focus on unexpected behavior

Specify Level of Abstraction For analysis

Telemetry **Server**

- **Who does socket communicate with?**
- **Is communication according to specification?**

Dyn-SAVE Capabilities (Vision)

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

What components in the client are affected by unspecified communication?

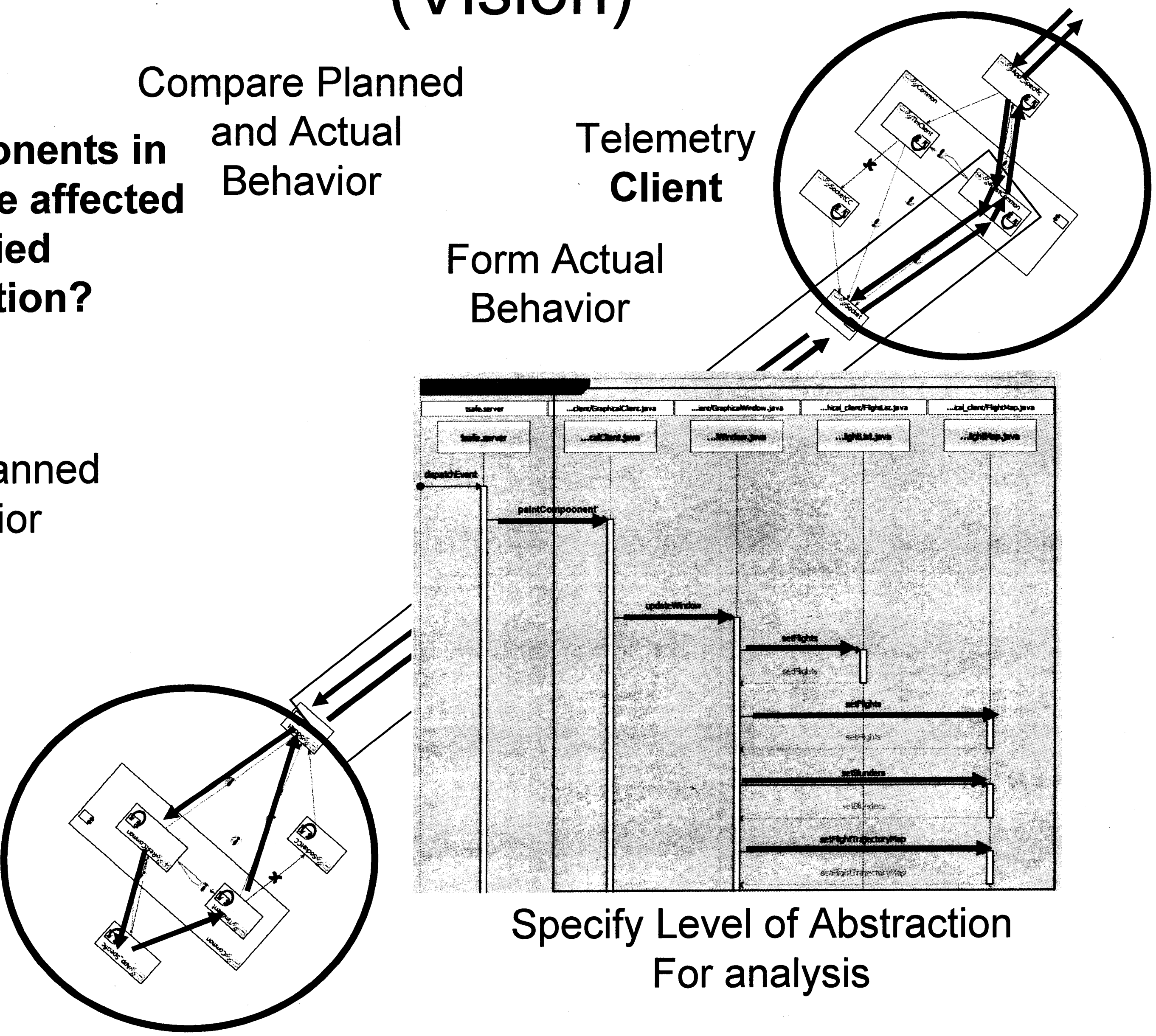
Compare Planned and Actual Behavior

Telemetry **Client**

Form Actual Behavior

Reuse Planned Behavior

Telemetry **Server**



Specify Level of Abstraction For analysis

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Feasibility Study
Deviations from Interface Control Documents

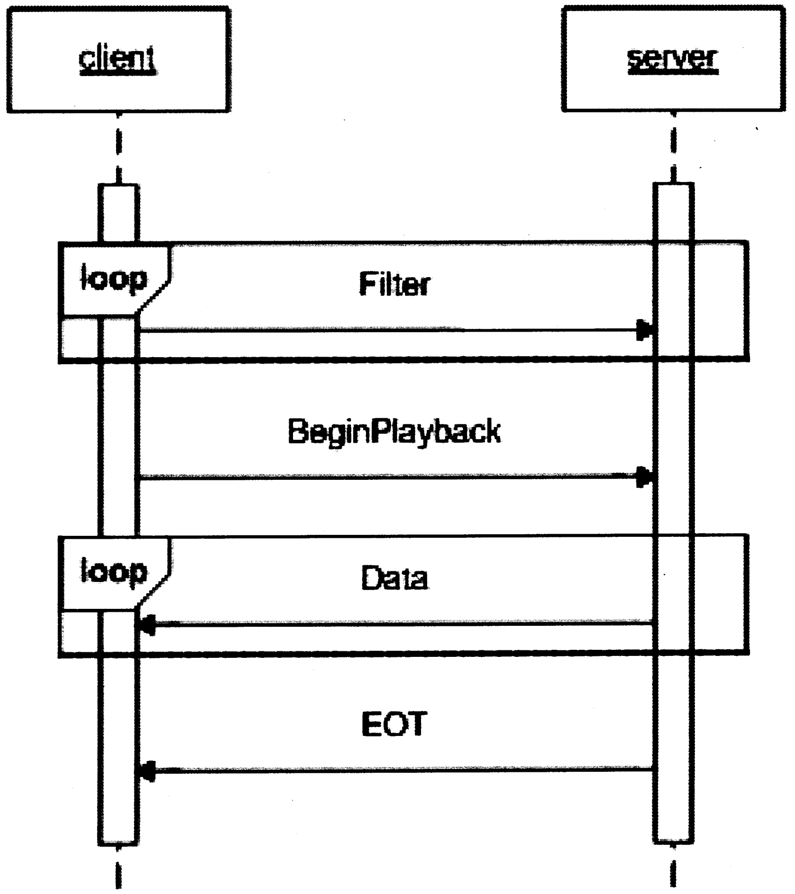
- One organization develops Ground Systems
  - Think of it as a server
- Other organizations develop clients
- Everybody follows the same ICD (Interface Control Document), but interpret them differently
- The systems are never tested together until they are made operational
- The result is subtle deviations from specified behavior that are difficult to spot
- Need a way to specify expected behavior and compare to actual

Research Questions

- Would it be possible to model the communication as a sequence diagram and use it to detect deviations?
- Can we identify a way to do iterative modeling, i.e. start with abstract models and add more details as necessary?
- How would we visualize deviations?
- Would such an approach be practical?

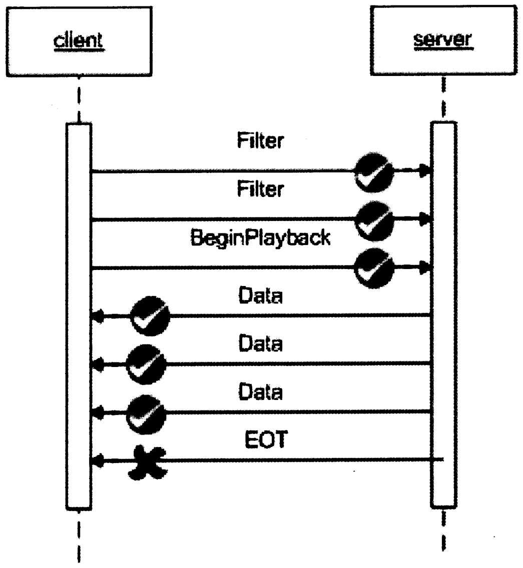
Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

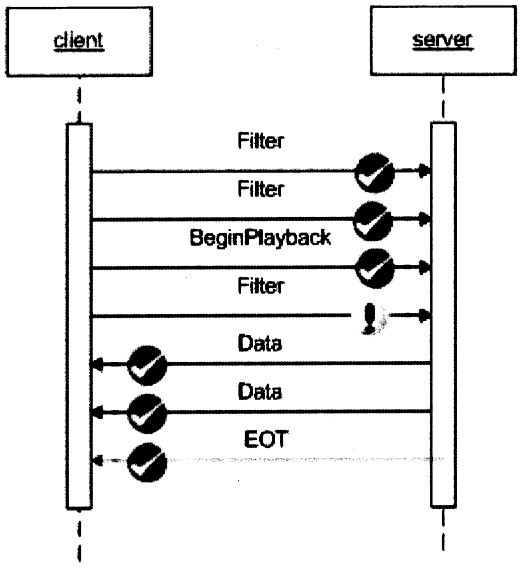
High level planned sequence diagram



Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Evaluation based on high level planned sequence diagram

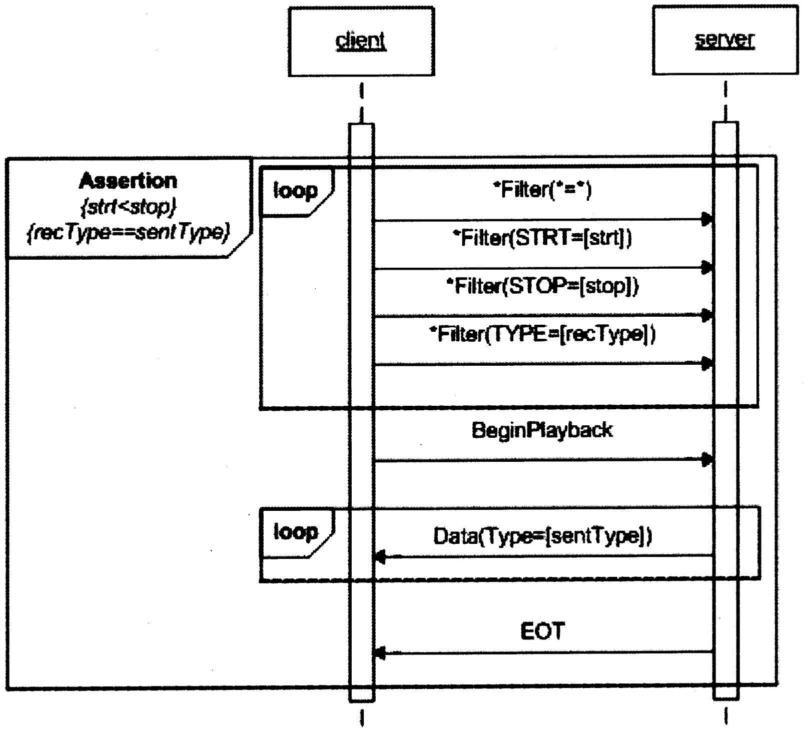




Left: EOT is missing from “correct sequence”. Right: An illegal extra filter is sent after BeginPlayback message has been sent.

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Low level planned sequence diagram

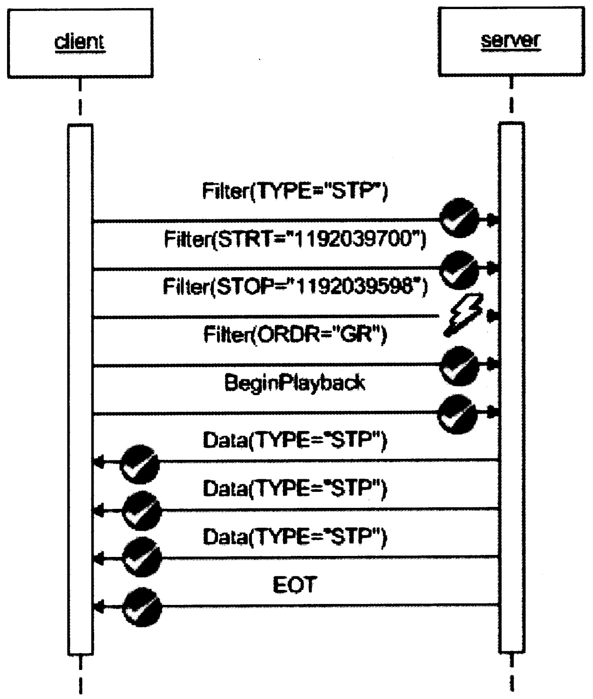


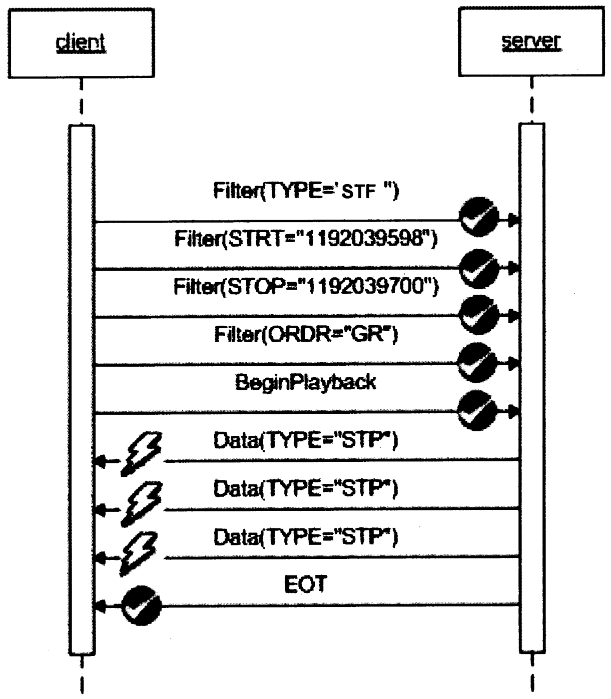
Rules:

1. Start time must be less that stop time
2. The data type of each of the received data messages must be the same as the specified type.

Mechanisms: assertions and message parameters
Assertions evaluated for each message

Evaluation based on low level planned sequence diagram





Left: Stop time < Start time. Right: STF ordered – STP received.

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Observations

- Sequencing problems
  - Messages that occur unexpectedly or out of order, can be detected by using high level sequence diagrams without details
- Content problems
  - Problems related to the content of messages rather than to the order of messages require a more detailed modeling approach
- The suggested approach
  - Is based on assertions in combination with sequence diagrams
  - Seems to be a feasible approach for this problem.

Research Approach

- Work as one team with problem-owners at APL
- Experiment with technology; apply to our testbed
- Evaluate technology; apply it to APL's CGS
- Improve technology based on feedback, results
- Repeat

- Extend to NASA projects
  - e.g. through the Research Infusion program

Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Summary

- Our approach is to apply Visualization and Evaluation concepts to Dynamic Analysis
- Applying Static SAVE ideas to dynamic data:
  - Planned vs. Actual
  - Version n vs. Version n+1
  - Abstractions as well as Details
- R&D driven by real needs from Ground and Flight
  - Using the TSAFE testbed for experimentation
  - Evaluating together with APL in their context
- Tech transfer through Research Infusion projects