A Data Abstraction Architecture for Mission Operations

Scott Bell*, David Kortenkamp*, Jack Zaientz**

*TRACLabs Inc., Houston TX USA
e-mail: [korten,scott]@traclabs.com

**Soar Technologies Inc., Ann Arbor MI, USA
e-mail: jzaientz@soartech.com

Abstract

Spacecraft generate huge amounts of data. For example, the International Space Station (ISS) has over 250,000 individually identified pieces of low-level telemetry and commands. Newer space vehicles and habitats will still generate incredible amounts of telemetry. People have difficulty interpreting large streams of numeric values because our brains are optimized for visual and symbolic reasoning activities. While computers are good at interpreting large streams of numeric values, advanced automation software is similar to humans in being optimized for symbolic reasoning. Thus, a significant problem in both monitoring and controlling space vehicles, robots and habitats is turning large streams of numeric values into more abstract and more symbolic information. This paper describes a Data Abstraction Architecture that formalizes the data abstraction process for space systems.

1 Introduction

Modern space systems such as satellites, human spacecraft, planetary probes and space robots are highly sensored and generate large amounts of data. For this data to be useful to humans monitoring these systems and to automated algorithms controlling these systems, it will need to be converted into more abstract data. This abstracted data will reflect the trends, states, and characteristics of the systems and their environments. Currently this data abstraction process is manual, ad hoc, and intermingled with control systems. It is manual in the sense that either humans do the abstraction in their heads or the data abstraction is done by hand-coding computer programs for each data item. It is ad hoc in the sense that each data abstraction is developed on its own with no representation of how it relates to the tasks being performed or to other data abstractions. It is intermingled with the control systems in that data abstractions are irreducible and difficult for other programs, like displays and analysis tools, to access. In this paper we describe the Data Abstraction Architecture (DAA) that allows engineers to design software processes that iteratively convert spacecraft data into higher and higher levels of abstraction. The DAA provides a canonical way to assemble and interact with data abstraction. Similar to control architectures (e.g., [2, 9], a data abstraction architecture provides a tool-box of components and connections that allow engineers to build and maintain data abstraction systems.

Our architecture consists of the following key components:

- Data abstractors: A defined transformation of data signals from one form to another, usually more abstracted or specialized, form.
- Sensor Event Abstraction Language (SEAL): An XML scheme that formally represents the connections between different data abstractors and the incoming data.
- Data Abstraction Reasoning Engine (DARE): Encodes the SEAL-represented data abstraction architecture in a computer program that is connected to the data stream, runs in real time and produces outputs for higher-level control systems, system engineers or crew.
- Development environment: An end-user oriented software tool to aid in the construction of DAAs and the export of them to DARE.

Taken together these components provide a mechanism for representing and accessing the data necessary to monitor and control space vehicles, habitats and robots.

2 Data Abstractors

One of the principal functions of the data abstraction architecture is to define the operations that may be performed on an input message stream. Whether performed on the message content or the message envelope, these are referred to as abstractors, for their resulting product is an abstraction of the input data that is consumed by the next abstractor in the graph. Different classes of abstractors focus on different stages in the transformation process represented by a Data Abstraction Network (DAN). A DAN

is a directed graph ordering of the data source, abstractors, and sink that represents the order of transformations to make on the data events. Below is a representative selection of the available abstractors:

Message Management abstractors are designed to help manage the message flow of the data bus:

- Sampler reduces the number of messages handled to a manageable subset and are generally used at the beginning of a DAN.
- Temporal Alignment ensures that messages coming from different sensors at different rates are grouped to represent events occurring at the same time.

Data Manipulation abstractors focus on transforming the sensor data itself:

- Mathematical Functions (Ratio, Average, Arithmetic) specify math expressions to perform on the input values
- Unit Transformation changes units of measure

Output Management abstractors focus on which results are to be included in the output, and how they will appear:

- Categorical Binner groups numeric values into symbolic categories (e.g., "low, "med, "high) and display only the symbolic value.
- Trim drops values that lie outside specified ranges.

Each of these is described in more detail in the following subsections.

2.1 Message management

Message management abstractors help organize the continuous flow of information into the data abstraction network. The following message management abstractors have been developed.

Temporal Alignment Collects a single message from each of multiple input streams and outputs a single new message that contains the set of collected messages. Incoming messages (or sub-properties of message) are mapped into individual properties on a single outgoing message. The abstractor will support different triggering rules including "all new events received, "one new event received, or "new event from input X received." As, by definition, event streams work at different rates and send events in non-deterministic orders, the abstractor must have buyer management rules that determine how to handle the case when multiple events arrive on one input while waiting for an event on a different input. These rules are similar to the Sampler abstractors, and will include last value, first value, and mean. Temporal Alignment is the only Message Management abstractor that is required in every DAN.

Sampler Used to accumulate a buffer of discrete messages over a defined observation period and reduce them to a single message. Sampler takes a single input stream of messages and allows every Nth message, as determined by the' sample-rate' property, to pass through. Example sample abstractors may include 'last value,' 'first value,' and 'mean.'

Accumulator Collects events from a single event stream over a defined observation period (in terms of time, message number, or external signal) and releases a single output message that contains an ordered set of the messages collected. The accumulator takes a single field from the input message and adds a new property to the message which is a list of the last 'buffer-size' values of that field.

2.2 Data manipulation

Data manipulation abstractors combine multiple pieces of data (or the same data over a time interval) to create a new piece of data. The following data manipulation abstractors have been developed.

Ratio Takes a ratio between two numeric values in the incoming message and stores it in the 'result' property on the output message.

Average Outputs the average of the input message values.

Arithmetic Performs the specified mathematical functions on the input data and outputs the result.

Equivalence Compares two or more values from an input message, and outputs a value of "true if they are equal within some threshold and "alse if not.

Count Counts the number of homogenous values in the input message and outputs the number.

Outliers Checks an array of values for outliers using an interpercentile range. The final output is an array of the outlier values.

2.3 Output management

Output management data abstractors organize the data before it is output to a human (via a display) or to an external algorithm. The following output management abstractors have been implemented.

Categorical Binner Reduces real values into symbolic categories and outputs the symbolic value. For example, a temperature reading could be transformed into "high," "medium, or "low." The symbol output map for each bin is defined in the expressions property.

Trim Selects a sub-element of an input message and outputs a new message with only that sub-element. The location of the sub-element in the output message may be specified by the 'result' property.

Conditional Propagation Performs a test on a value in a message and then conditionally propagates the message if that test resolves to true. Tests may be logical or arithmetic. Nothing is added by this abstractor.

Propagate on Changes Watches for a change in the input value within a series of received messages then passes the same message if the value changes. Nothing is added by this abstractor.

Conditional : Forwards data based on whether a Boolean expression evaluates to TRUE or FALSE.

Sinks Specifies the output target for the transformed data. This typically represents the intended output from the DAN.

3 Sensor Event Abstraction Language

The Sensor Event Abstraction Language (SEAL) is an XML grammar that defines data manipulation and message handling operators, enabling the description of sophisticated transformations on event-based telemetry data. The SEAL syntax and semantics are intended to support the computational requirements of NASA telemetry and telemetry management processes and align to the conceptual model of those processes held by expert NASA flight control engineers. Finally, the language is intended to support rapid visual development and inspection of data transformation by skilled engineers who are typically trained in disciplines other than software engineering. Typical data transformation programming environments, such as discrete event simulations, circuit design simulators, spreadsheets, and test systems, structure data transformation using a graph representation. The standard version of graph semantics adopted by these environments typically makes a number of assumptions about node and edge structure of the graph, specifically that nodes represent information processing functions that input a single type of information and output a single value. The edge represents the current output of the node. The placing of a new value on an input edge causes the adjacent nodes to fire, causing an update on their output edges. This in turn causes new node firings until the graph reaches quiescence. In such a graph, where all edges represent a current value, the graph itself can be seen as having a global memory state. This structure works fine in environments where input singles are single-data type and where data delivery is relatively guaranteed. Neither of these constraints is appropriate in the telemetry environment where messages often have composite structure and can have significant latency and drop effects. This structure also does not easily support the annotation of node-outputs with meta-data such as error conditions, pedigree descriptions, or uncertainty values. SEAL, while also using a graph-based structure, draws on an event-based message passing semantics similar to that found in enterprise messaging systems [3] In this semantics, an edge represents the path a message may follow, but not the message itself. Along these edges, there are two classes of nodes: message-element operators, which transform data elements in a message into a new data element appended to the message, and message-envelope operators, which manipulate message structure to route or merge messages or to remove data elements from a message. In this semantics, a message is a complex memory structure while the overall graph has no persistent state. As messages traverse the graph they pick up new data elements, building up not only an output value but a processing history. This conceptually clarifies and computationally simplifies standard telemetry processes such as sampling and temporal alignment (to manage data over/under runs). Sampling, for example, can be instantiated as a message-envelope operator, accumulating a set of messages over a period of time into a single message, followed by message-data operator (e.g., mean) that reduces data values in the individual message sections into a single data value. Temporally aligning temperature readings from different thermal sensors in a spacecraft chamber is a matter of linking each of their message-paths to a message-envelop operator which groups them by time range. This message-passing semantics is well suited to the NASA telemetry environment, matching how flight controllers understand telemetry processing and the kinds of configurations they would expect to perform. It also allows a number of computational benefits including easy distributed processing and load-balancing due to the lack of global memory and easy integration with messaging systems. Finally, it provides an excellent basis for feeding data into high-level controllers due to its ability to output both raw instrument data and processed or symbolized information in the same message.

This basis allows for more complicated structures to be built up. For example, a Quiescence Filter only passes a value out the far side if that value has remained within tolerances for a prescribed time interval. Figure 1 shows one possible implementation of a Quiescence Filter. First an alignment operator and equivalence operator pair gathers a set of messages together (from different data sources)

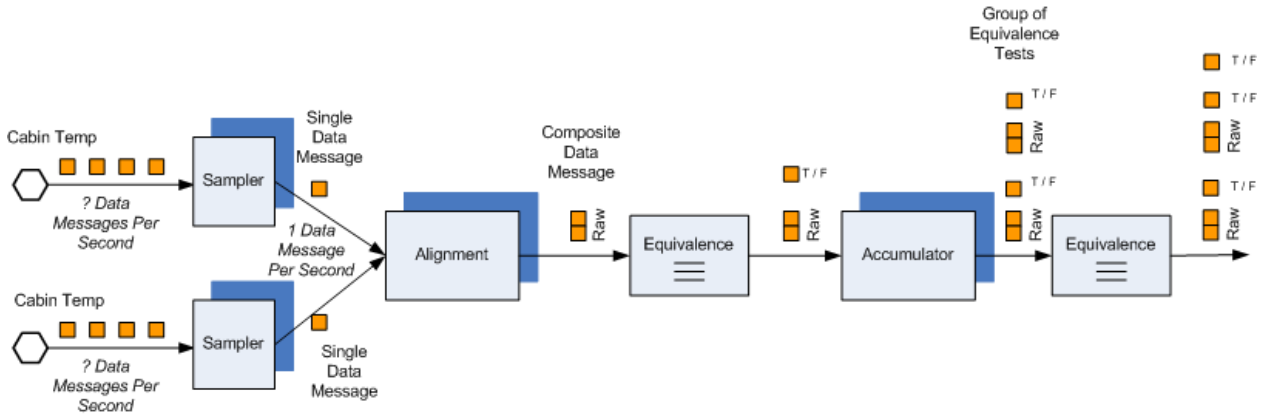

Figure 1. An example SEAL abstraction architecture describing a quiescence filter.

and evaluates them to see if they are within tolerance. Second, the message, which contains all the original messages and the output of the equivalence test, is passed to another accumulator operator and equivalence operator pair. This pair compares whether the group that is within tolerance has remained in tolerance for the prescribed amount of time. Note that this example makes strategic use of both message-envelop operators (temporal alignment and accumulation) and a simple message-data operator (equivalence) to instantiate the more complex notion of quiescence. The formal specification of the language semantics is a specialization and XML rendering of the general Set-Function (SF) syntax described by Bertziss [1] Our version streamlines the general SF grammar, reducing expressiveness in favor of non-software engineer programmability. Where SF allows the description of rich preconditions to trigger each processing event, we restrict this to a simple message existence / location test requiring preconditions that discriminate based on message content to be constructed in a separate processing event. Like SF we divide event processing into two functions, one that reduces a data set to an output value (or set of values) and one that positions the output of the first function at some location in the output message. We support the visual selection of set functions by selection of message-data or message-envelope operator objects in the visual editor, and the linkages of these functions to event-preconditions, by visually linking abstractor objects via message-paths. Currently the function for placing data output in a message object is not handled visually, but through the textual specification of a message path. These features ensure that, like SF, SEAL is a general language that can construct a large set of possible data transformation graphs and do so in a manner that is primarily visual.

4 Data Abstraction Reasoning Engine and SEAL editor

The Data Abstraction Reasoning Engine (DARE) takes a SEAL file and instantiates it as a Java program that is connected to the data sources and sinks, runs in real time, and produces events for higher-level control systems, system operators, or crew. DARE uses ActiveMQ and Java Messaging Service (JMS) as middleware to allow for distributed execution of SEAL files. DARE is light-weight and portable and can be run on several current operating systems including Windows, Mac OSX and Linux.

The SEAL editor allows the user to graphically build a Data Abstraction Network (DAN) from a list of abstractors provided by a library file. The SEAL editor was developed as an Eclipse plug-in. Eclipse is an open-source integrated development environment (IDE) framework that provides a platform of pre-existing functionality, which allows developers to create language-specific IDEs without re-writing everything from scratch. The SEAL editor allows for a drag-and-drop approach to building Data Abstraction Networks and doesn't require the end user to understand XML or SEAL.

5 A Data Abstraction Example

We tested the Data Abstraction Architecture by implementing an International Space Station (ISS) flight rule as a Data Abstraction Network. Flight rules govern the operation of space vehicles. Flight rules are currently written in Microsoft Word and are not monitored by software systems. The flight rule we chose governs the operation of the smoke detectors on ISS. The flight rule determines when a smoke detector is 'dirty' and must be serviced. It does this by looking at specific telemetry coming from that smoke detector, performing calculations on that teleme-

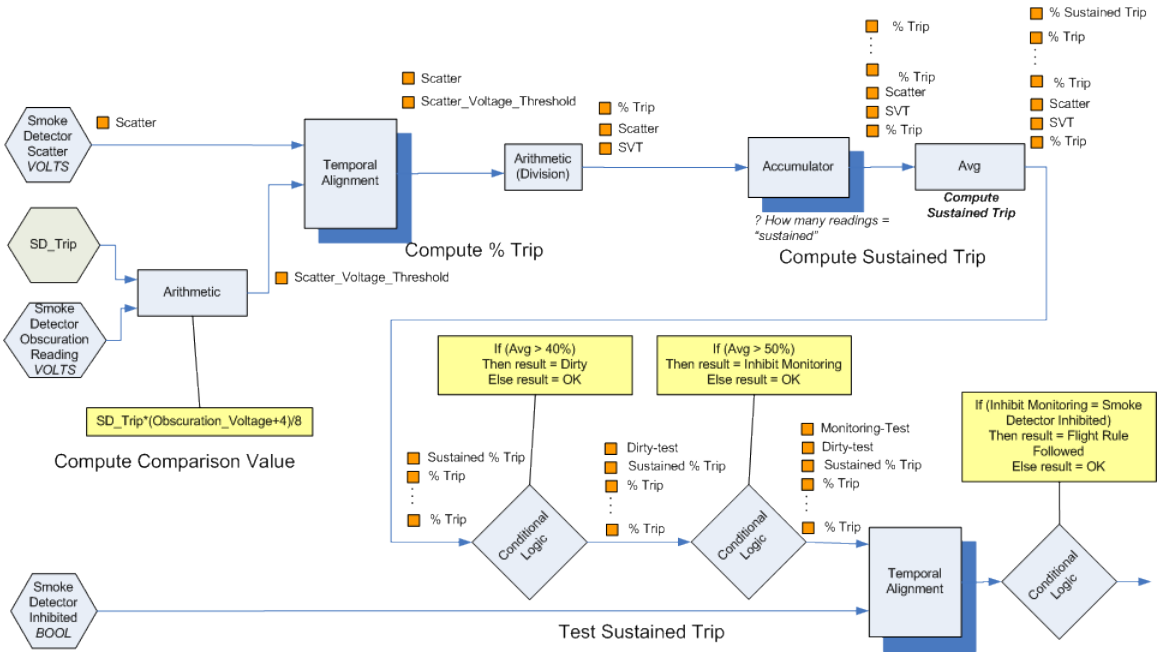

Figure 2. A data abstraction network for the smoke detector flight rule.

try, and comparing those calculation to pre-defined limits. We manually translated this flight rule into the SEAL language. A visual representation of this SEAL file, including all of the abstractors, is shown in Figure 2.

Starting in the upper left, the hexagons are two sensor inputs from ISS – the scatter voltage and the obscuration voltage. The other hexagon is a constant, determined experimentally and referenced in the flight rule. An arithmetic abstractor computes a new value from the obscuration voltage and the constant. Next, a temporal alignment abstractor makes sure that we are comparing two values that were obtained at the same time. This prevents stale values from being compared to new values. This abstractor outputs two values that are temporally consistent. Another arithmetic data abstractor computes the percent tripped for that smoke detector. An accumulator abstractor gathers up those readings over time (the "sustained" part of the flight rule) and passes all of those to an average abstractor. That average is first compared to the limit 40% and then to the limit 50% with either the result of OK or dirty and inhibit monitoring respectively. That is, if the limit is less than 40% the smoke detector is OK. If the limit is between 40% and 50% the smoke detector is dirty and must be serviced. If the average is greater than 50% the smoke detector must be inhibited (turned off). Finally, telemetry that states whether or not the smoke detector is inhibited is compared to the result of the computation to determine whether the smoke detector should be inhibited. If they agree, then the flight rule is being followed. If not, then the flight rule is being violated.

5.1 Flight controller interaction with flight rules

The Mission Control Technology (MCT) project is developing new display software for NASA Mission Control Center (MCC) [13]. We developed a custom MCT component that connects with DARE to retrieve information. We also created a view of that component so that users can inspect the data abstraction network and see the abstracted data. The view allows a user to see the entire data abstraction network in graphical form (see Figure 3). It also allows the user to inspect the values, units, etc. of any part of the data abstraction network as well as change data abstractor parameters.

5.2 Connecting to ISS data

We tested our data abstraction network against actual ISS smoke detector telemetry. We did this by connecting DARE to the Information Sharing Protocol (ISP) network that publishes live ISS telemetry. We accessed this network through a Virtual Private Network (VPN) account with NASA JSC. The data abstraction network was successfully able to monitor a specific smoke detector on ISS. ISP also has the capability to playback a hand-built file of telemetry values. We used this to test the data abstraction network against data that violated the flight rule. This test confirmed that the data abstraction network could detect when the flight rule was being violated. This proof-

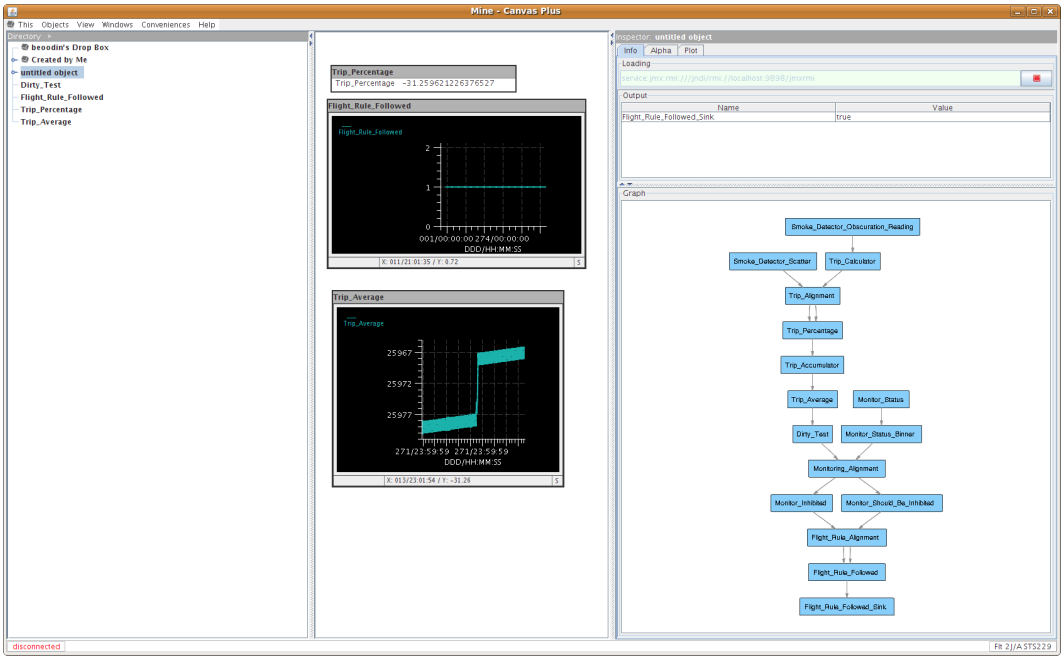

Figure 3. Screen shot of the MCT user interface for the data abstraction network.

of-concept demonstration was done using the MCT mission control interface. Figure 4 shows the set of processes used for this proof-of-concept demonstration. The XTCE referred to in the diagram stands for the XML Telemetric and Commanding Exchange XML schema. XTCE is an emerging standard for representing telemetry and commands for space systems. It has been adopted as a standard by NASA to represent commands and telemetry. XTCE serves as the source of our raw telemetry information. We can also output an XTCE file that contains the abstracted telemetry generated by DARE.

6 Related work

Several autonomous control architectures had explicit data abstraction. One clear example is the Supervenience architecture [12]. The architecture consisted of communicating levels in which lower levels pass data about the world to higher levels. At the same time higher levels pass goals down to lower levels. It is implemented using a blackboard architecture at each level. Each level also contains its own uniform data representation. Several agent-based systems have looked at the information retrieval and integration problem (see [5] and [8]). Some early examples of agent systems for information retrieval and coordination include COLLAGEN [10], Infomaster [4] and work by Jennings [6] and Lesser [7]. Another example would be the Mobile Agents work of Clancey and Sierhuis, especially with respect to robotics interaction [11].

7 Conclusions

Data abstraction is a critical component of the space system information flow. Dealing with raw telemetry is difficult for human controllers and for automated systems. This paper describes a formal methodology for abstracting spacecraft telemetry into more abstracted data constructs. This methodology includes a abstraction representation, an abstraction engine and an integrated development environment. The architecture was validated using live ISS telemetry running in real time. The architectural components have been integrated with the next generation mission control software tool suite.

8 Acknowledgments

This work funded under NASA contract NNX08CA11C. The authors wish to thank Jeremy Frank and Jay Trimble of NASA Ames Research Center and Alan Crocker and Christie Bertels of NASA Johnson Space Center for their contributions to the ideas presented in this paper.

References

[1] Alfs Berztiss. Formal specification methods and visualization. In Shi Kuo Chang, editor, *Principles of Visual Programming Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1990.

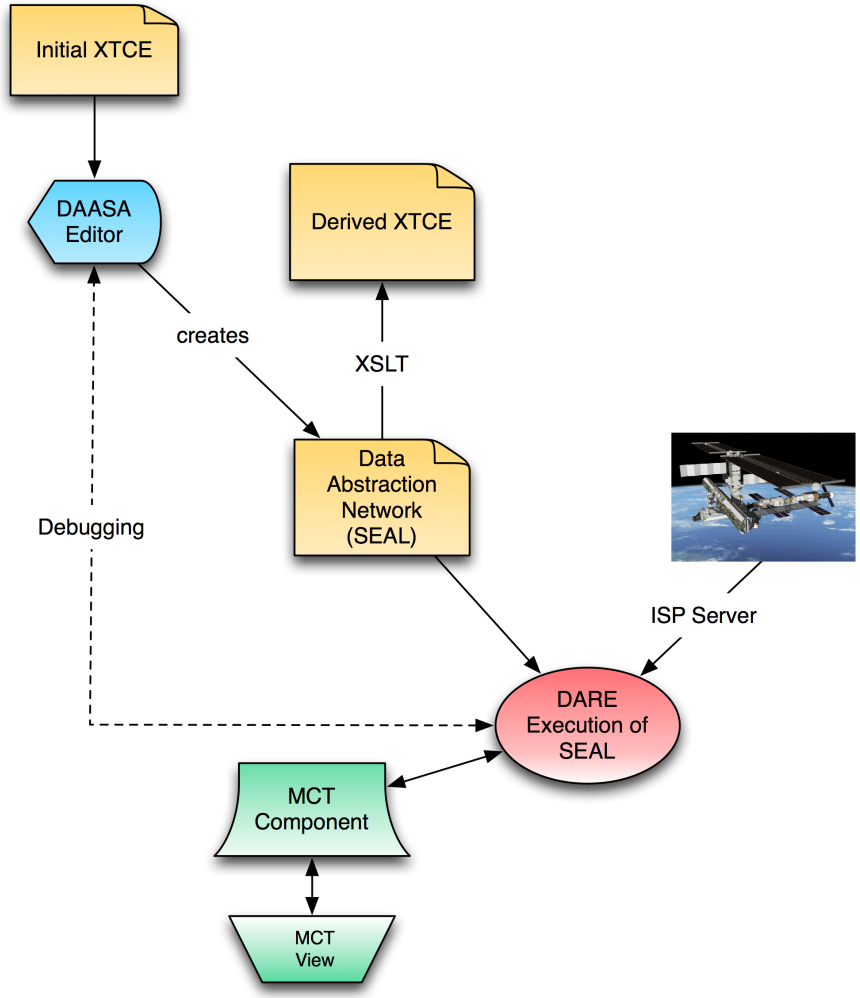

Figure 4. The processes that were used to monitor ISS flight rules.

[2] R. Peter Bonasso, R. J. Firby, E. Gat, David Kortenkamp, David P. Miller, and Marc Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1), 1997.

[3] Dave Chappel. *Enterprise Service Bus*. OReilly Media, Inc., Sebastopol, CA, 2004.

[4] Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: an information integration system. In *Proceedings of the ACM SIGMOD Conference*, pages 539–542, 1997.

[5] D. S. Haverkamp and S. Gauch. Intelligent information agents: Review and challenges for distributed information sources. *Journal of the American Society for Information Science*, 49(4), 1998.

[6] N. R. Jennings. Coordination techniques for distributed artificial intelligence. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence, Sixth-Generation Computer Technology Series,*. John Wiley and Sons, New York, 1996.

[7] V. R. Lesser. Reflections on the nature of multi-agent coordination and its implications for an agent architecture. In *Autonomous Agents and Multi-Agent Systems (AAMAS-98)*, 1998.

[8] T. W. Malone, K. Y. Lai, and K. R. Grant. Agents for information sharing and coordination: A history and some reflections. In J. M. Bradshaw, editor, *Software Agents*. AAAI Press, Menlo Park CA, 1997.

[9] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1), 1998.

[10] C. Rich and C. L. Sidner. COLLAGEN: a collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction*, 8(3-4), 1998.

[11] Maarten Sierhuis, Jeffrey M. Bradshaw, Alessandro Acquisti, Ron van Hoof, Renia Jeffers, and Andrzej Uszok. Human-agent teamwork and adjustable autonomy in practice. In *Proceedings of the 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space: i-SAIRAS 2003*, 2003.

[12] L. Spector and J. Hendler. Planning and reacting across supervenient levels of representation. *International Journal of Intelligent and Cooperative Information Systems*, 1(3), 1992.

[13] Jay Trimble, Joan Walton, and Harry Sadler. Mission control technologies: A new way of designing and evolving mission systems. In *AIAA Space Operations 2006*, 2006.