Data Abstraction Architecture for Spacecraft Autonomy

Scott Bell and David Kortenkamp
TRACLabs Inc.
1012 Hercules
Houston Texas 77058
korten@traclabs.com

Jack Zaientz
Soar Technology Inc.
3600 Green Court, Suite 600
Ann Arbor, MI 48105

Spacecrafts generate huge amounts of data. A significant challenge for autonomous control systems and human operators is ensuring that the right data (and combinations of data) are available at the right time for control and decision-making and ensuring that the data is at the right abstraction level. In this paper, we describe a data abstraction architecture that provides a canonical way to assemble and interact with data abstractions. We have developed an open, flexible toolkit that allows end users to build data abstraction networks. Two use cases were successfully tested against a Lunar habitat simulation, demonstrating that high level state information can be generated by the data abstraction architecture and used by a high level controller. Our approach improves the process of both control and monitoring of space systems by separating controls and displays from data abstraction.

I. 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 present the Data Abstraction Architecture for Spacecraft Autonomy (DAASA) that allows engineers to design software processes that iteratively convert spacecraft data into higher and higher levels of abstraction. DAASA also formalizes the relationships between data and control and the relationships between the data themselves.

II. Related Work

Several autonomous control architectures had explicit data abstraction. One clear example is the Supervenience architecture.[1] 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. The Reactive Action Packages System (RAPS)[2] had its own data abstraction component that was added in the early 90s[3] . This functioned more as a conceptual network in which data could be represented at many levels. It was primarily used as a way to communicate with humans. RAPS also led to a pattern recognition architecture called the Complex Event Recognition Architecture (CERA).[3] It was primarily concerned with expressing parsers that would recognize complex patterns in streams of data. The Open System Architecture for Condition Based Maintenance organization (OSA-CBM) provides a hierarchical breakdown of information for use in condition-based maintenance. It denes software modules, data structures, and the interfaces and protocols between the modules that make up a system. It encompasses a range of functions from sensing hardware through diagnosis and prognosis all the way to presentation of the diagnosis with recommended maintenance actions. It has its own interface language called the Abstract Interface Denition Language (AIDL), which is very similar to a CORBA IDL. In fact, translators from AIDL to IDL exist. This architecture is focused on system health management and maintenance and does not support more traditional autonomous control activities. However, it is being accepted more and more widely in industry and government.

III. Overview of DAASA

DAASA allows engineers to design software processes (called data abstraction networks) that iteratively convert sensor data into higher and higher levels of abstraction. DAASA is a series of mathematical or logical transformations of sensor data to provide appropriate abstractions that can be used on-board to control the vehicle or that can be transmitted off-board to a control station for human monitoring. DAASA formalizes the relationships between data and control and the relationships between the data themselves. Thus, DAASA provides a canonical way to assemble and interact with data abstraction. Similar to control architectures (e.g., 3T[4] and Remote Agent[5]) DAASA provides a tool-box of components and connections that allows engineers to build and maintain data abstraction systems.

III.A. DAASA Components

DAASA consists of several integrated components and representations. These include:

Data events Define the data upon which DAASA operates, including telemetry, derived data, symbols and triggers. Events are heterogeneous, hierarchical, multi-value messages and may occur asynchronously. DAASA defines an XML schema for data events.

Data source The producer of data events in DAASA. This generator may include either raw telemetry data events generated by hardware sensors or preprocessed data events from a low-level controller or other abstraction architectures. Typically events are generated on change of value or from sampling of the underlying hardware.

Data sink The consumer of a data event outputs from the DAASA. This receiver may include high-level control systems, crew displays, logging or maintenance systems, or other abstraction architectures.

Data abstractors Define either a transformation, caching, or reorganization of data events, typically for a more abstract or specialized form. Abstractors consume data events coming from data sources or other abstractors and produce events to sinks or other abstractors.

Sensor event abstraction language (SEAL) An XML grammar which defines the data abstractors, the abstractor's message handling operations, and the directed graph connecting the abstractors.

Data abstraction reasoning engine (DARE) Instantiates a SEAL file in a computer 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.

Development environment An end-user oriented software tool to aid in the construction, debugging and viewing of SEAL files.

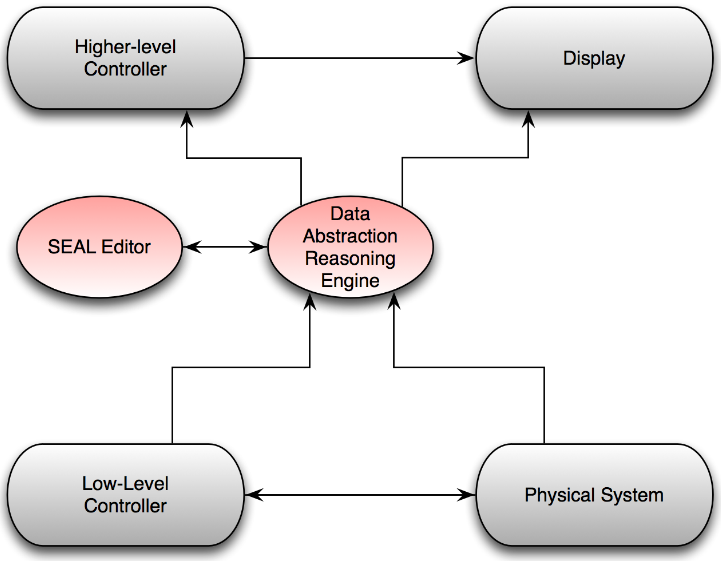


Figure 1. General overview of the Data Abstraction Architecture for Spacecraft Autonomy

Taken together these components provide a mechanism for representing and accessing the data necessary to monitor and control Constellation vehicles, habitats and robots. Figure III.A shows a high-level overview of how DAASA integrates with other space vehicle systems. The physical system in the lower right provides raw sensor telemetry into the Data Abstraction Reasoning Engine (DARE). Low level controllers also use this same, unabstracted data and can produce their own events for DARE. These two systems are data sources. The SEAL editor creates an XML file that defines the data abstraction network, which is read in by DARE. The abstracted data can be used by displays intended for human consumption or by higher-level controllers. Both are data sinks.

III.B. Data Abstractors

There are a large number of data abstraction algorithms that might be useful to a DAASA designer. These include:

- **Unit transformations** that convert an input value in an event into another input value.
- **Rate computations** that analyze the rate of change of an input value in an event.
- **Outliers** that take a set of homogenous values in an event and use interpercentile ranges to find values that are different from the others.
- **Equivalences** take a set of homogeneous values in an event and creates a new boolean value of **true** if the values are the same, **false** if not.
- **Samplers** are any abstractor that can be used to accumulate a buffer of discrete messages over a defined observation period and reduce them to a single message. Example sample abstractors may include ‘last value,’ ‘first value,’ and ‘mean.’
- **Accumulators** collect events from a single event stream over a defined observation period (in terms of time, message number, or an external signal) and then releases a single output message that contains an ordered set of the messages collected.
- **Conditional Propogators** perform a test on a value in a message and then passes on the message if that test resolves to **true**. Tests may be logical or arithmetic.
- **Propagate on Changers** takes an input value in a message and fires the same message if the value changes.

- **Temporal Alignments** collect a single message from each of multiple input streams and outputs a single new message that contains the set of collected messages. 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 buffer 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 above, and will include "last value," "first value," and "mean".
- **Categorical Binners** reduce real valued events into symbolic categories. For example a temperature input could be reduced to a "high", "medium", or 'low" output event by a categorical binner.
- **Reducing Abstractors (Mean, Median, Count, Sum, Last, First, Max, Min)**: take a homogeneous set of data elements (as assembled by an accumulator) and reduces the set to a single element. In the cases of mean, sum, and count, this value of this element is computed from the homogeneous set. In the case of last, first, max, min, and median, it is a member of the set.
- **Trims**: take an input of a single message and outputs a new message with fewer data elements as selected by the user.
- **Mathematical functions** that allow for specifying a math expression (e.g., $input1 + input2$) that is computed on the inputs in the event.
- **Logical functions** that allow for specifying a logical expression (e.g., $input1 \vee input2$) that is computed on the inputs in an event.

We have build a subset of these to implement the use cases described in Section VI. A generic DARE API allows for easy creation of new abstractors by experienced programmers.

IV. Sensor Event Abstraction Language (SEAL)

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.

IV.A. Description

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.[6] 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.

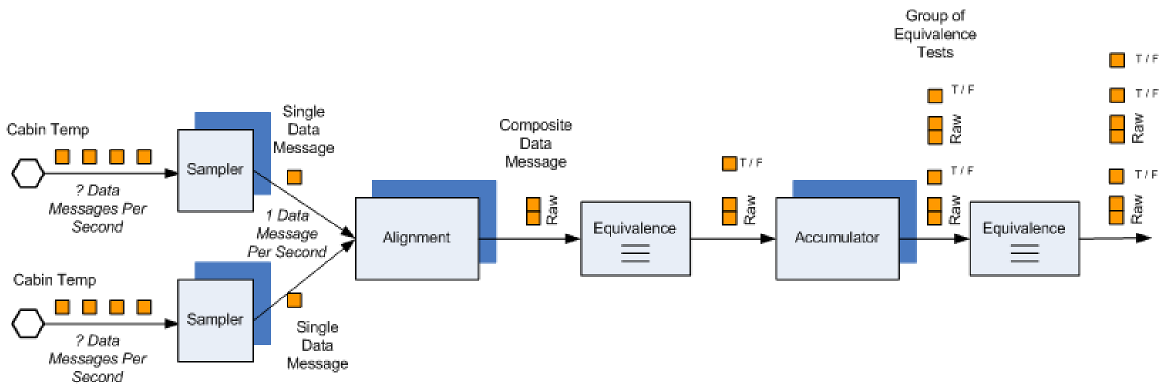
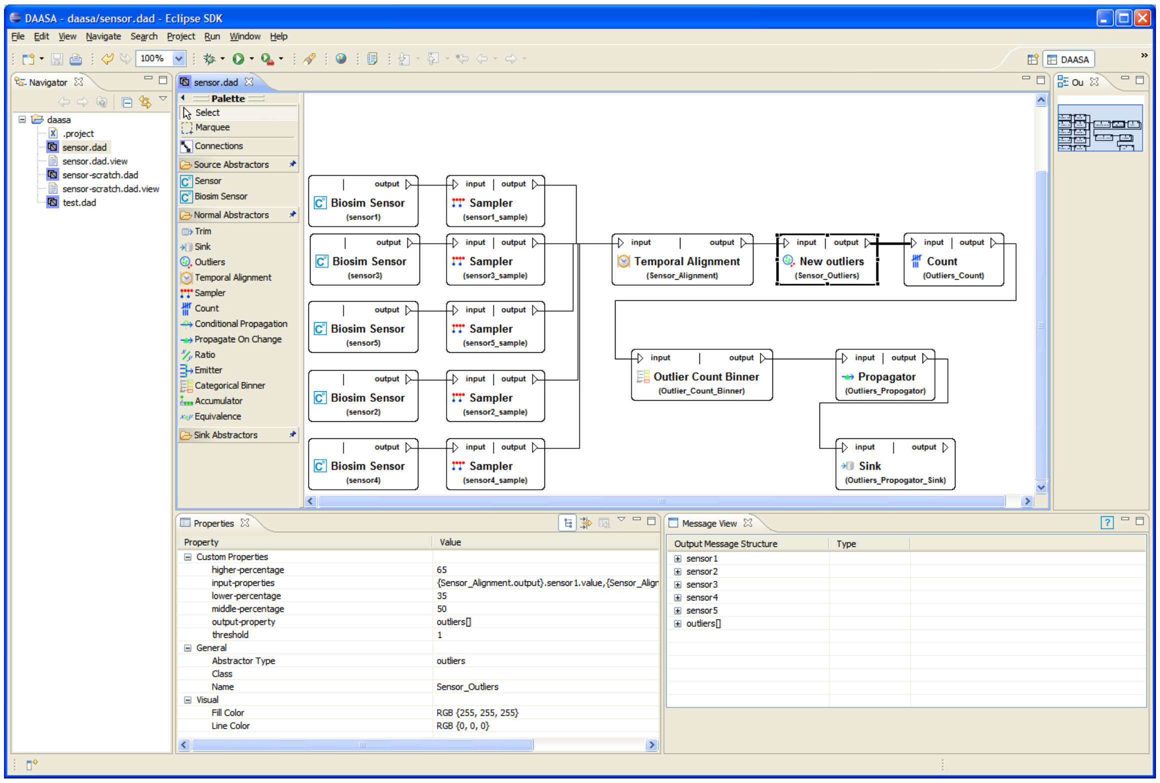


Figure 2. Quiescence Filter

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 with tolerances for a prescribed time interval. Figure 2 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) 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 with-in tolerance has remained in tolerance for the prescribed amount of time. Not 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.[7] 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.

IV.B. SEAL Visual Editing Environment



Property	Value
Custom Properties	
higher-percentage	65
input-properties	{Sensor_Alignment.output}.sensor1.value,{Sensor_Align
lower-percentage	35
middle-percentage	50
output-property	outliers[]
threshold	1
General	
Abstractor Type	outliers
Class	
Name	Sensor_Outliers
Visual	
Fill Color	RGB {255, 255, 255}
Line Color	RGB {0, 0, 0}

Output Message Structure	Type
sensor1	
sensor2	
sensor3	
sensor4	
sensor5	
outliers[]	

Figure 3. SEAL visual editing environment

The SEAL visual editing environment (shown in Figure 3) has been developed in Eclipse, a Java open source editing platform and provides the expected basic functionality including drag and drop placement of operators, automatic routing of message-path lines, local save and load and static validation of SEAL expressions, connection with the DARE engine for run-time debugging including remote start and stop, variable-watches, and breakpoints. To support NASA telemetry applications, the editor natively supports the XML Telemetric and Command Exchange (XTCE) standard descriptions and identifiers for telemetry data sources. To support the event-passing semantics of the language, we implemented an algorithm that computes the current message structure at any point along the message graph. This algorithm enables users to view and easily manipulate the message structure when assigning message-data operators to data values in a message and when assigning message data-operator output to specific locations in a message.

V. Data Abstraction Reasoning Engine (DARE)

The Data Abstraction Reasoning Engine or DARE, is a distributed, message based software program that takes as input a SEAL file, instantiates the listed abstractors, connects the abstractors to each other, the sources, and the sinks. When DARE is finished initializing, data sources are producing events from live data, abstractors are computing on those generated events, and sinks are consuming the resultant events.

V.A. Implementation

Sinks are implemented as formalized end-points of the abstraction network, though third party applications may access any of the intermediate events. These intermediate events are provided externally to allow transparency to DARE's event processing. DARE is implemented in Java using ActiveMQ as the messaging broker. Messages passed between source, sinks, and abstractors are accessible using a variety of protocols including JMS, JMX, Openwire, REST, XMMP, and CORBA. DARE can be remotely debugged using the SEAL visual editing environment using JMX.

As previously noted, data events passed between abstractors, sinks, and sources are heterogeneous, hierarchical, multi-value messages. To support this, DARE implements events as messages with any number of properties. A message property is a name/value pair where the value may be an atomic value (string, number, etc), an ordered list of values, or another message. In a single message, each property is unique and may appear only once.

Consider an output message from an abstractor of type Counter called SensorCounter:

```
{
   sensors: [
   {
      name: cabin pressure sensor,
      units: kPA,
      value: 101
   },
   {
      name: airlock pressure sensor,
      units: kPA,
      value: 85
   }
   count: 2
}
```

In this example, there are two top level properties: sensors and count. Count has a simple value of the integer 2. Sensors has the complex value of a list, each element of the list containing three more properties: name, units, and value.

V.B. SEAL Addressing Scheme

To do any work on input messages, DARE abstractors need an addressing scheme to find the data in the message on which to process. DARE uniquely address a property in a message using the following:

Source abstractor: This is the name of the source abstractor

Property path: A sequence of message property names.

List indexing: For list values, a unique, zero-based integer indicating the index of the desired value.

Aggregate Operator: For list values, indicates all values of the list, rather than a single value.

These five concepts taken together uniquely address the field of a message coming from a particular abstractor output. We encode this address as follows:

```
{<abstractor name>}.<property path>
```

The first two elements (abstractor name and first property name) are always required. Consider another abstractor of type Average called SensorAverage. If SensorAverage wanted to compute a mean on the received message, it would use the following two input addresses:

```
{SensorCounter}.sensors[0].value, {SensorCounter}.sensors[1].value
```

This would locate the two values 28 and 101 respectively. We could also have SensorAverage use the list aggregate operator to find its values thusly:

```
{SensorCounter}.sensors[].value
```

Note the empty "[]" on contacts. This indicates we would like to operate on all values in the contacts list. To create an output, the same addressing scheme is used, except the list aggregate operator is disallowed. For example, SensorAverage needs a new property to place the result of its processing. The output address:

```
{SensorAverage}.average
```

would create a new property in the message called "average" and place the average of the sensor values therein. The output message of SensorAverage would look like this:

```
{
   sensors: [
   {
      name: cabin pressure sensor,
      units: kPA,
      value: 28
   },
   {
      name: airlock pressure sensor,
      units: kPA,
      value: 101
   }
   count: 2
   average: 64.5
}
```

VI. Use Cases

We examined two separate use cases for DAASA. Both use cases received sensor data from a life support simulation called BioSim. BioSim is a dynamic system simulation tool developed by NASA Johnson Space Center over the past decade.[8] Mathematical models for typical components found in various life support systems are fully integrated and highly congurable. Simulation progresses in hourly time increments, with each unit process producing and consuming various resources in designated stores. An XML conguration file containing the design of the system initializes the simulation including settings such as random failure and stochastic performance. BioSim has been successfully used and veried in many life support optimal design applications, including reliability analysis, control system testing, and power system design verification. The configuration chosen for the use cases is based on a lunar mission containing one cabin, one crew member, an airlock, and abundant food, water, and oxygen.

VI.A. Detecting a Malfunctioning Sensor

In the first use case, we created an abstraction network to monitor five sensors that measured the carbon dioxide in the crew cabin. Nominally, the sensors should all be reporting the same value (aside from a bit of noise). However, we planned to fail one sensor and have DAASA report the failing sensor immediately. The network we created is shown in Figure 4.

First, each carbon dioxide sensor is sampled to 1Hz. This means each Sampler abstractor is generating one event every 1 second. These Sampler events are all consumed by the Temporal Alignment abstractor. This abstractor was configured to collect the Sampler events until one event from each Sampler had arrived. When this happens, a new event was published by Temporal Alignment containing the list of events from each Sampler. The Outliers abstractor would process this list, looking at each sensor reading for an anomalous sensor reading. If one is found, it is added to a list of outliers. If not, an empty list is passed. The Outliers fires a new message as soon as it's able to process its input. Count takes the event from Outliers and determines the length of the outliers list in its input event. Count fires a new event as soon as its able to determine this, which is sent to the Propagate On Change abstractor. If the count value in the message has changed, a new event is sent to the display. If not, Propagate On Change discards the event. To implement this network, we started with the SEAL editor as shown in Figure 5. Figure 6 shows the sensor values and the event detected by DARE.

VI.B. Controlling an Airlock

For the second use case, we created an abstraction network that was used by a high-level controller to manage an airlock for an EVA. The controller uses DARE to signal it when high-level goals have been accomplished. The procedure for readying the airlock for EVA is as follows:

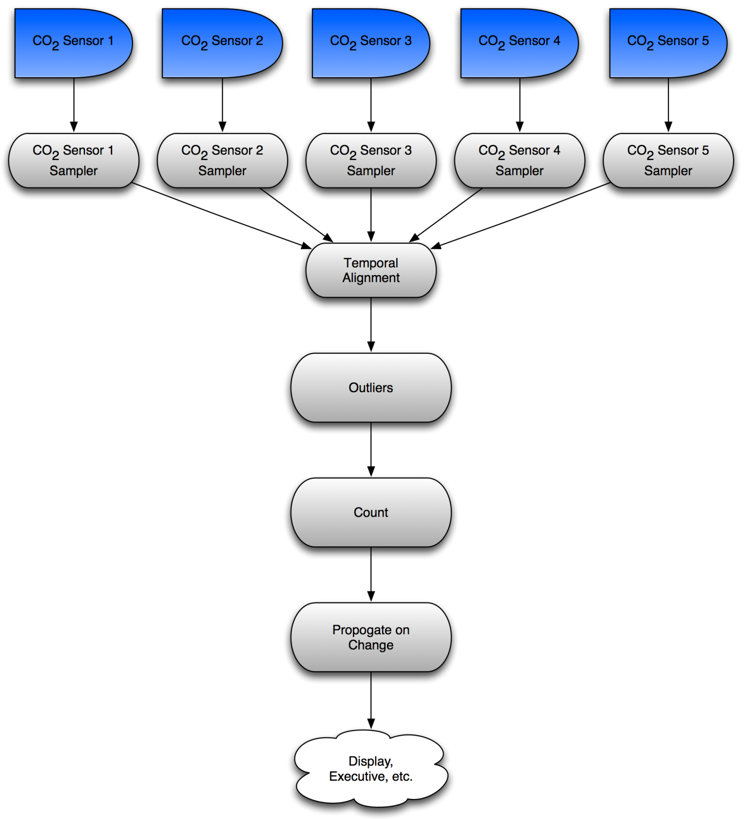
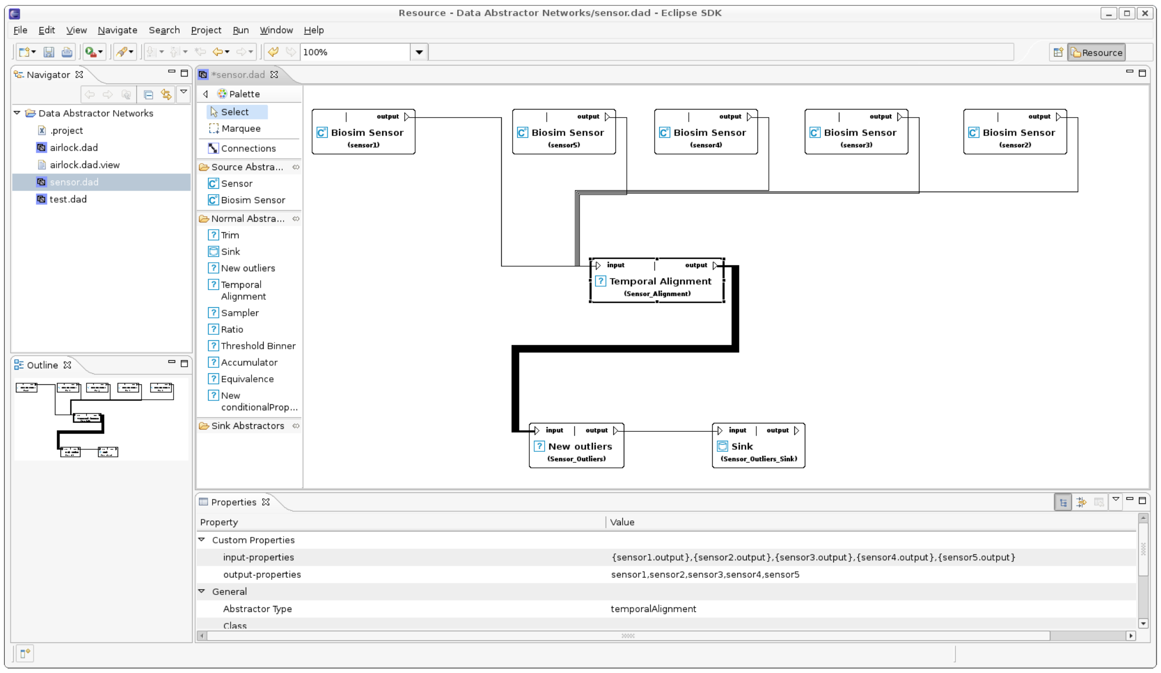
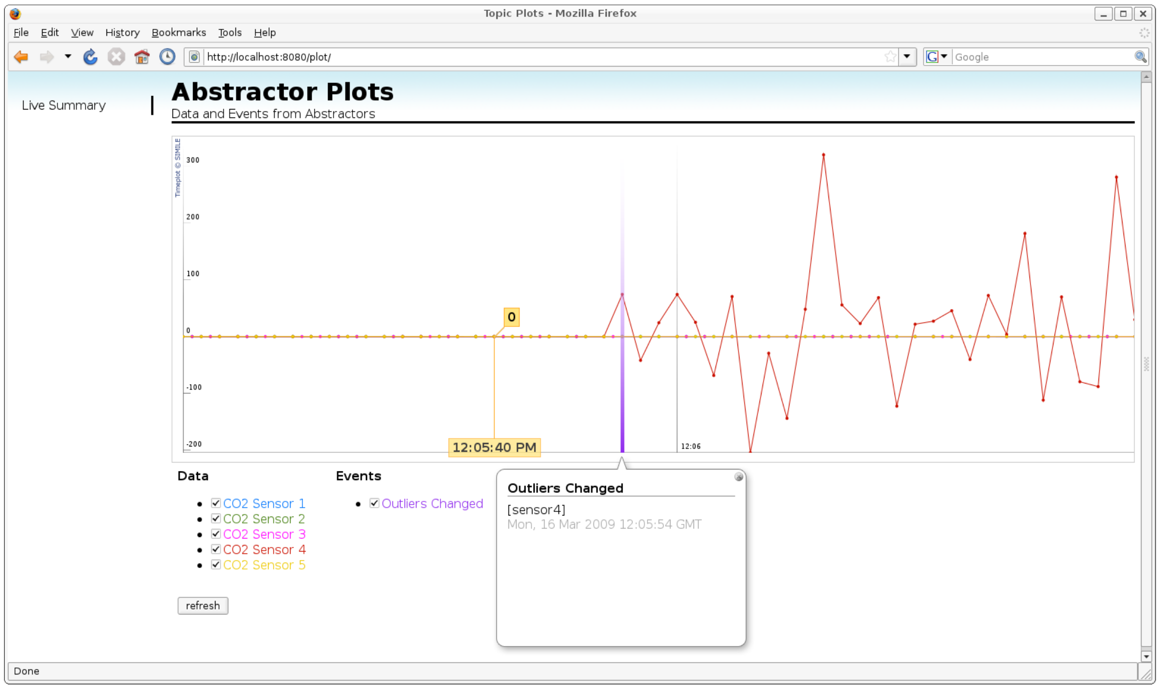


Figure 4. A data abstraction network for detecting a carbon dioxide sensor failure.



Property	Value
Custom Properties	
input-properties	{sensor1.output},{sensor2.output},{sensor3.output},{sensor4.output},{sensor5.output}
output-properties	sensor1,sensor2,sensor3,sensor4,sensor5
General	
Abstractor Type	temporalAlignment
Class	

Figure 5. The carbon dioxide sensor failure network being built in the SEAL editor.



Outliers Changed
[sensor4]
Mon, 16 Mar 2009 12:05:54 GMT

Figure 6. Graph of events and data during the sensor failure

1. Get the airlock's total pressure and oxygen level to NORMAL
2. Get the oxygen level to PREBREATHE
3. Wait three hours
4. Reduce the total pressure to VACUUM

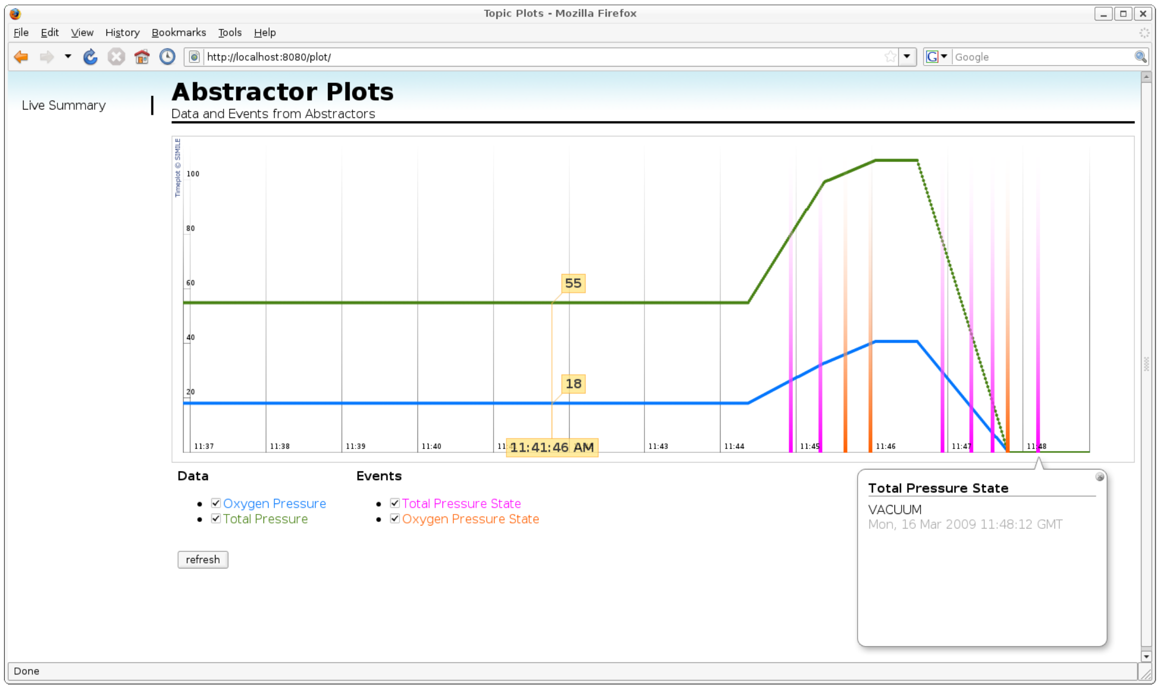
The data abstraction network determined the airlock state (e.g., NORMAL, PREBREATH, VACUUM, etc.) by sampling the total airlock pressure and the partial pressure of oxygen. It then used several categorical binners to output a symbolic state. A quiescence abstractor was used to ensure that the system was in a stable state before the high-level controller took another action. The high-level controller adjusted actuators in BioSim to accomplish goals and used outputs from DARE for state estimation. In Figure 7 shows the various states the airlock undergoes in readying for EVA and how they correspond to the total airlock pressure and the partial pressure of oxygen.

VII. Future work

Our next steps include adding the ability to create composite abstractors, that is, to be able to build abstractors of abstractors. This will make it significantly easier to build complicated abstraction networks. We are also planning to connect the data abstraction architecture to a real telemetry stream coming from a NASA vehicle (either live or recorded).

In the future we plan on formalizing SEAL language semantics relative to both concurrent programming languages such as Hoares Communicating Sequential Processes[9] visual process descriptions such as Berztiss SF.[7] We also plan to revisite SEAL syntax to improve our ability to compose larger abstraction blocks from primative operators and to improve the temporal model used by the primative operators to cover a wider set of NASA telemetry management problems.

We are exploring uses of the data abstraction architecture in NASA's Mission Control Center (MCC). The current method of doing data abstraction in mission operations is to write special software called "computations" (or "comps") that take in a few values of raw telemetry and create a new telemetry value that is added to the telemetry stream. Comps are not an ideal solution for several reasons. First, they require



Total Pressure State

VACUUM

Mon, 16 Mar 2009 11:48:12 GMT

Figure 7. Graph of events and data during the airlock's readying for EVA

software programming skills on the part of the operator (or reliance upon software programmers). Second, there is often a significant delay between recognizing the need for a comp and its instantiation. Finally, comps only convert numeric values into other numeric values and only occur at the lowest level of the data stream. This last drawback prevents the creation of higher and higher levels of data abstraction that all feed one another. Our approach improves the process by allowing "comps" to be built and evaluated on-the-fly and in a formal manner. We are integrating our software with NASA's Mission Control Technologies (MCT) program, which is developing the next generation of MCC displays.[10]

VIII. Conclusions

We have designed and implemented a prototype data abstraction architecture and used it in several simple scenarios. The data abstraction architecture and its associated SEAL grammar formalizes the transformation of data from raw sensory telemetry to higher-level data. Such abstractions are critical in monitoring and controlling complicated space systems. By standardizing the representations and processes we are creating a toolkit that engineers can use to build data abstractions. Initial conversations with NASA flight controllers and control engineers has revealed a growing need for architectures such as the one presented in this paper.

References

[1]Spector, L. and Hendler, J., "Planning and Reacting across Supervenient Levels of Representation," *International Journal of Intelligent and Cooperative Information Systems*, Vol. 1, No. 3, 1992.

[2]Firby, R. J., "An investigation into reactive planning in complex domains," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1987.

[3]Fitzgerald, W., Firby, R. J., Phillips, A., and Kairys, J., "Complex event pattern recognition for long-term system monitoring," *Proceedings of the AAAI 2003 Spring Symposium on Human Interaction with Autonomous Systems in Complex Environments (available from AAAI Press at www.aaai.org)*, 2003.

[4]Bonasso, R. P., Firby, R. J., Gat, E., Kortenkamp, D., Miller, D. P., and Slack, M., "Experiences with an Architecture for Intelligent, Reactive Agents," *Journal of Experimental and Theoretical Artificial Intelligence*, Vol. 9, No. 1, 1997.

[5]Muscettola, N., Nayak, P. P., Pell, B., and Williams, B. C., "Remote Agent: to boldly go where no AI system has gone before," *Artificial Intelligence*, Vol. 103, No. 1, 1998.

[6]Chappel, D., *Enterprise Service Bus*, OReilly Media, Inc., Sebastopol, CA, 2004.

[7]Berztiss, A., "Formal Specification Methods and Visualization," *Principles of Visual Programming Systems*, edited by

S. K. Chang, Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1990.

[8]Kortenkamp, D. and Bell, S., "Simulating Advanced Life Support Systems for Integrated Controls Research," *Proceedings International Conference on Environmental Systems*, 2003.

[9]Roscoe, A., *The Theory and Practice of Concurrency*, Prentice Hall, Upper Saddle River, New Jersey, 1997.

[10]Trimble, J., Walton, J., and Sadler, H., "Mission Control Technologies: A new way of designing and evolving mission systems," *AIAA Space Operations 2006*, 2006.