09ICES-0239

Data Abstraction Architecture for Monitoring and Control of Lunar Habitats

Scott Bell and David Kortenkamp
TRACLabs Inc, Houston TX 77058

Jack Zaientz
Soar Technologies Inc., Ann Arbor MI 8105

ABSTRACT

A Lunar habitat will be highly sensored and generate large amounts of data or telemetry. 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 and *ad hoc*. We are developing a Data Abstraction Architecture (DAA) that allows engineers to design software processes that iteratively convert habitat data into higher and higher levels of abstraction. The DAA is a series of mathematical or logical transformations of telemetry data to provide appropriate inputs from a hardware system to a hardware system controller, system engineer, or crew. The DAA also formalizes the relationships between data and control and the relationships between the data themselves. We have connected our Data Abstraction Architecture to a simulation of a Lunar habitat in order to test its ability to aid in the monitoring and control functions.

INTRODUCTION

Space systems are growing more and more complicated and containing more and more sensors. The large amount of data generated is overwhelming both ground controllers and automated control systems. Coping with this sensor data will require developing software systems that can iteratively convert raw sensor data into more meaningful, derived data that can be used by controllers, both human and automated. In this paper we describe a data abstraction architecture that allows engineers to design software processes that convert habitat data into higher levels of abstraction. First, we present the architecture including its components and representations. Then we present some use cases in which the architecture is used to monitor and control a Lunar habitat. We then describe our experimental environment including a Lunar habitat simulation and discuss the performance of the data abstraction architecture in implementing the use cases. Next we talk about related work in this area. Finally, we discuss future directions and conclusions.

DATA ABSTRACTION ARCHITECTURE

A data abstraction architecture (DAA) formalizes the relationship between raw telemetry and derived data. A set of integrated components and representations are part of the DAA, including:

- Data events define the data (both raw and derived) upon which the architecture operates. Data events can be generated by hardware or software and happen asynchronously.
- Data abstractors, which are software programs that perform a transformation on data events. They consume certain kinds of data events and produce different data events.
- Sensor event abstraction language (SEAL), which is an eXtensible Markup Language (XML) schema that defines a grammar for connecting data abstractors, events, sources and sinks.
- Data abstraction reasoning engine (DARE), which instantiates a SEAL file into a computer program that connects to data sources and data sinks to perform abstractions.
- Development environment, which is an end-user software tool to aid in the construction, debugging and viewing of SEAL files.

Together these components provide a mechanism for creating more abstract data from raw telemetry. Each of them will be discussed in the rest of this section.

DATA EVENTS

Data events define the data upon which the DAA operates. Events are heterogeneous, hierarchical, multi-values message and may occur asynchronously. They can be generated from sensors, controllers or from data abstractors. These generators are called data sources. Data events are not simply a number, but instead are complicated data structures that can contain attributes such as confidence, timing, processing history, counts and arrays. Here is an example data event for a Lunar habitat scenario

```
{
  sensors: [
  {
    name: cabin pressure sensor,
    units: kPA,
    value: 101
  }
  {
     name: airlock pressure sensor,
     units: kPA,
     value: 85
  }
  average: 93
}
```

In this case the data event contains two sensed values, their units and their average. This would be produced from an averaging data abstractor.

DATA ABSTRACTORS

Data abstractors transform data events by performing processing on them. There is no limit to the number or kinds of data abstractors that can be built. Each data abstractor is implemented in a common programming language such as Java. The abstractor must conform to an Application Programmers Interface (API) that allows it to receive and generated data events and to be controlled. We have implemented the following kinds of data abstractors:

- Reducing
    - Average: Collects a series of values and produces their average (choice of mean or median)
    - Min/Max: Collects a series of values and produces their minimum or maximum value
- Timing
    - Temporal Alignment: Ouputs a single new data event containing a set of events that are all temporally common.
    - Sampling: Accumulates a buffer of discrete events over a defined observation period and produces a single data event.
- Symbol Processing
    - Categorical Binner: Reduce real value data events into symbol categories. For example, a temperature input could be reduced to “high”, “medium” or “low”.
- Propogation
    - Propogate on Change: Only generates an output data event when the input data event has a different value from the previous event
    - Quiescence: Only generates a data event when the input data event has remained constant for a set period of time.
- Mathematical and Logical
    - Basic mathematical and logical operations

Several of these will be described in the use cases presented later in the paper.

SENSOR EVENT ABSTRACTION LANGUAGE

The Sensor Event Abstraction Language (SEAL) is an XML grammar that defines data abstraction networks. It encodes the data abstractors and their parameters, the data events, the data sources and sinks and the connections between all of them. It also defines the data events. 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. 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.

DATA ABSTRACTION REASONING ENGINE

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.

EDITING AND DEBUGGING SEAL

The SEAL visual editing environment has been developed in Eclipse, a Java open source editing platform and provides the expected basic functionality including drag and drop placement of abstractors, 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.

USE CASES

We examined two separate use cases for the data abstraction architecture. Both use cases receive sensor data from a simulated Lunar habitat. In order to understand the use cases we first describe the simulated lunar habitat.

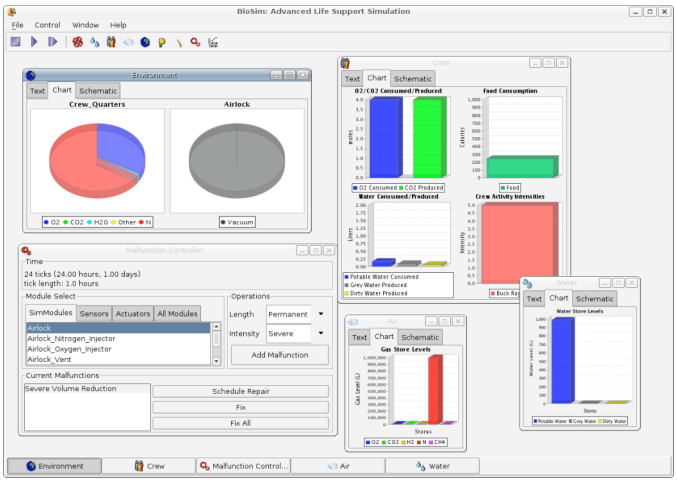

Figure 1: The BioSim habitat simulation

BIOSIM

The Lunar habitat ECLSS simulation is based on BioSim models developed over the past several years [4]. BioSim is a discrete-event simulation of a space habitat that models each of the life support components as processes that consume certain resources and produce other resources. For example, the air revitalization system model consumes air with a specific concentration of gases (e.g., high in CO_2 and low in O_2) and power and produces air with a different mixture of gases (e.g., low in CO_2 and high in O_2).

A simulated crew module is implemented using models described by [3]. The number, gender, age and weight of the crew are changeable. The crew cycles through activities such as sleep, maintenance, recreation, etc. During each activity they consume different levels of O_2 food and water and produce CO_2, dirty water and solid waste. The crew module is connected to a crew environment that contains an mixture of gases (an atmosphere) that they breathe. The initial size and gas composition (percentages of O_2, CO_2, H_2, O_2 and inert gases) are input parameters. As the simulation progresses the composition of gases in the atmosphere changes.

For this work, BioSIm was configured to contain a main crew cabin with a volume of 27 meters cubed and a connected airlock with a volume of 10 meters cubed. There are three controllers for the airlock. One controller pulls air out of the airlock and deposits it into the main crew cabin. Another controller puts air into the airlock from the main cabin. A third controller puts oxygen into the airlock from an oxygen store. Figure 1 shows the habitat simulation with a main cabin and an airlock (which is greyed out when at vacuum).

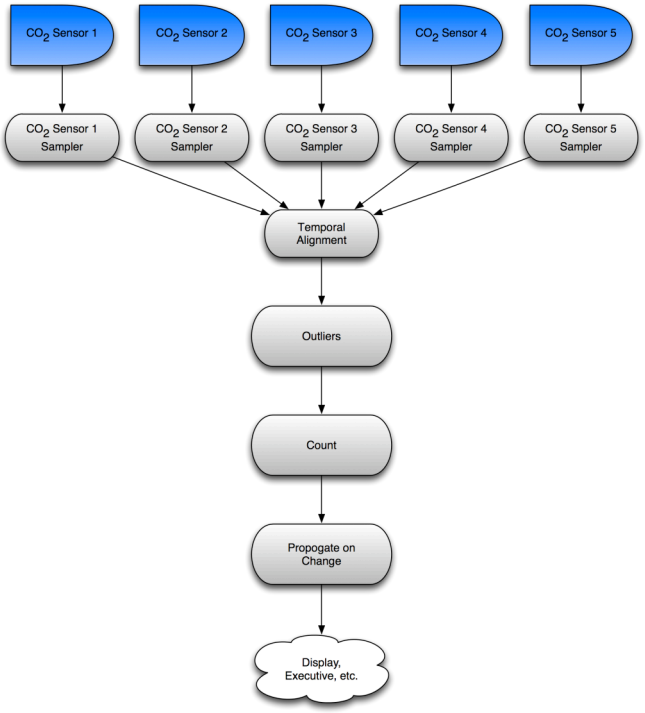

Figure 2: A data abstraction network for detecting a sensor failure

CARBON DIOXIDE SENSOR MONITORING

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 a DAA

detect and report the failing sensor immediately. The network we created is shown in Figure 2.

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.

HABITAT AIRLOCK CONTROL

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.

A data abstraction network was built to determine the airlock state by sampling the total airlock pressure and the partial pressure of oxygen. The airlock could be in one of several states:

- NORMAL: Airlock pressure and oxygen levels are the same as the crew cabin
- PREBREATHE: Airlock has a higher oxygen level
- VACUUM: Airlock has a very low pressure
- HIGH: Airlock has higher pressure than the crew cabin
- LOW: Airlock has a lower pressure than the crew cabin
- UNKNOWN: Airlock is not in one of the above states (usually because it is transitioning)

Several categorical binners are used 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. The data abstraction network for this use case is shown in Figure 3.

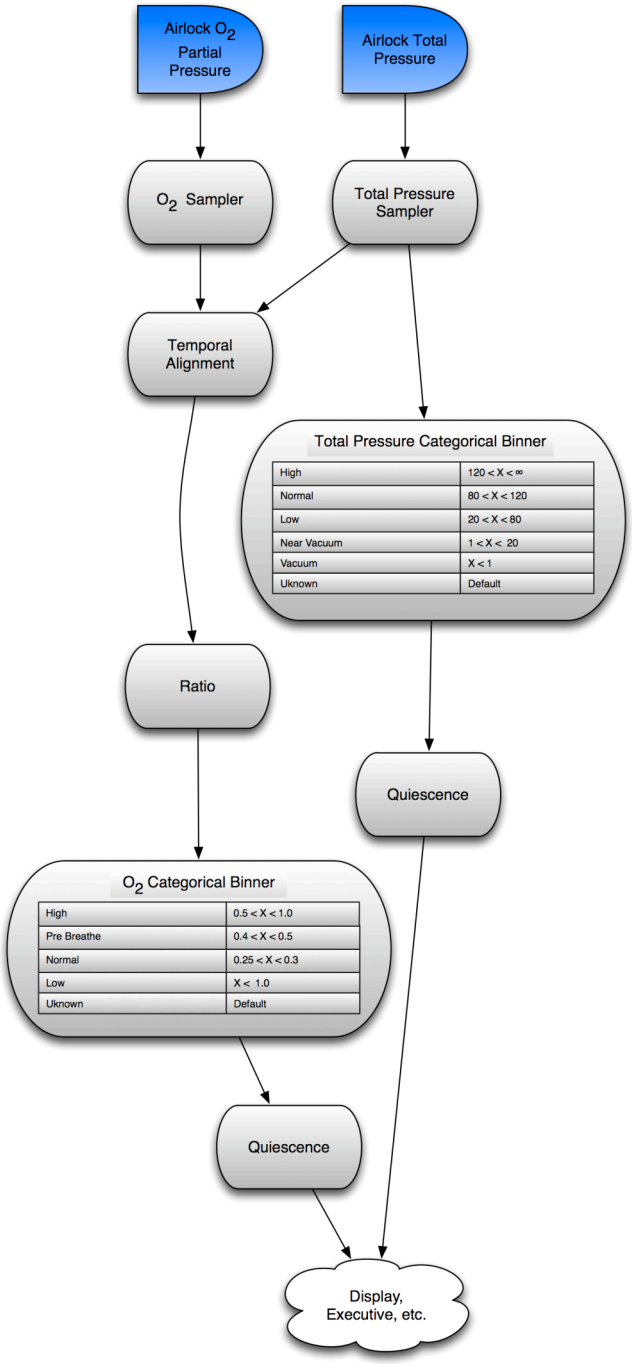

Figure 3: A data abstraction network for determining the airlock state

The high level controller executed a procedure written in the Procedure Representation Language (PRL), which is being used by NASA for the next generation of procedures for human space flight [5]. The procedure was as follows:

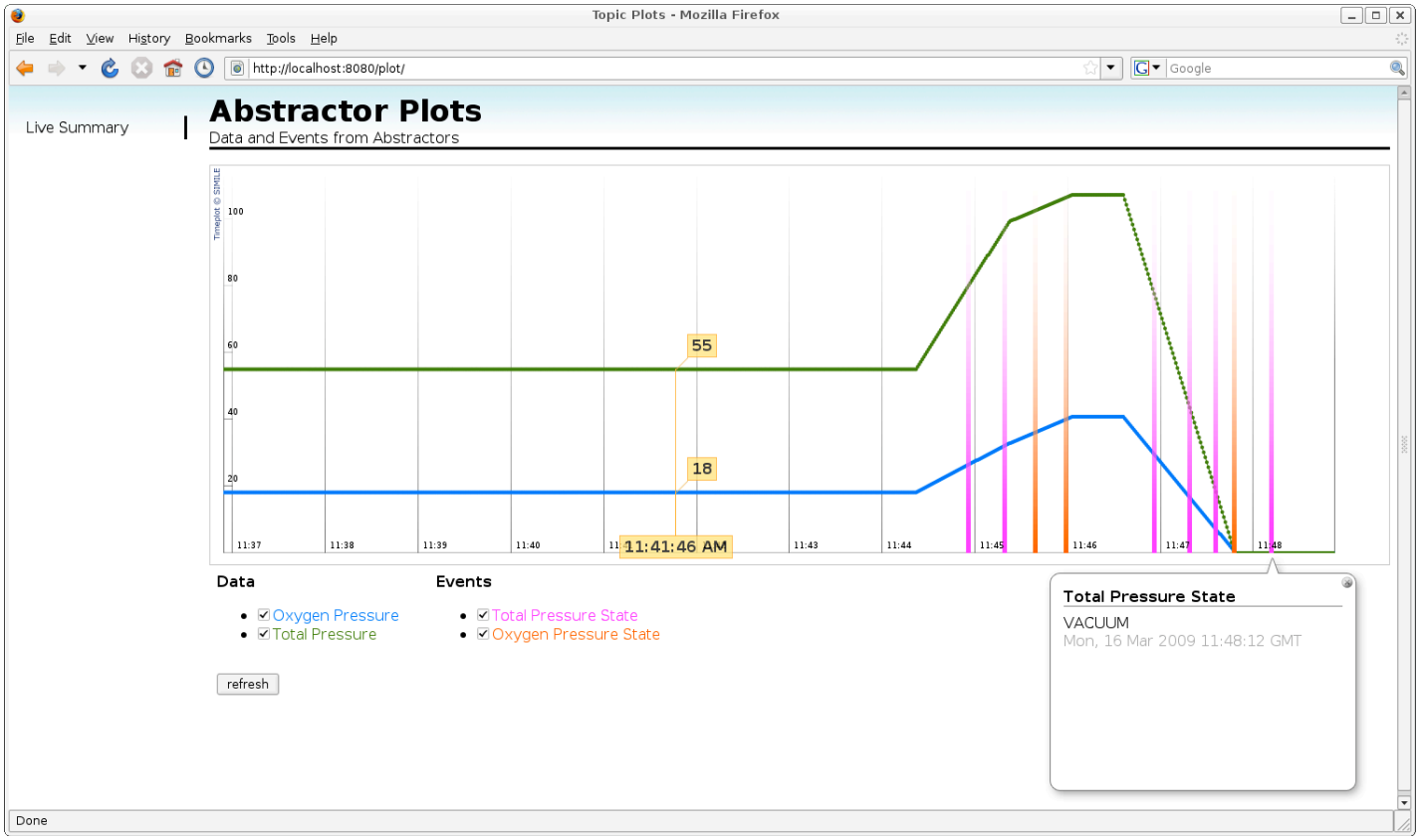

Figure 4: The output from the airlock state data abstraction network

- Get airlock's total pressure and oxygen level to NORMAL
- Get the oxygen level in the airlock to PREBREATHE
- Wait three hours
- Reduce the total pressure to VACUUM

In the use case, a human controller started execution of the procedure, which commanded the airlock controllers. The data abstraction architecture provided the high-level controller with state information.

Figure 4 shows a run of this use case. The pressure starts out low and is raised first to NORMAL, then to PREBREATHE, then held there for three hours and then the pressure is dropped to VACUUM. Each colored vertical line is the data abstraction network generating a new state for the airlock.

RELATED WORK

NASA flight controllers currently use an ad hoc "computations" or "comps" system to convert low-level telemetry into higher-level data. Comps are limited to single values and must be programmed by computer experts. NASA flight controllers also have a number of tools by which they can match telemetry against limits and show trends. These tools are difficult to change and limited to running on flight controller's computers only. Several autonomous control architectures had explicit data abstraction. One clear example is the Supervenience architecture [7]. 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) [1] had its own data abstraction component that was added in the early 90s [6]. 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) [2]. It was primarily concerned with expressing parsers that would recognize complex patterns in streams of data. In neither of these cases could novice programmers create data abstractions.

FUTURE WORK

We are continuing to develop the data abstraction architecture. We are implementing an ability to create composite abstractors. That is, allowing for the creation of abstractors of abstractors. This will make it significantly easier to build complicated abstraction networks.

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 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.

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.

ACKNOWLEDGMENTS

Several programmers at Soar Technologies were involved in the creation of the data abstraction architecture, including Kyle Aaron and Dave Ray. Jeremy Frank of NASA Ames Research Center provided significant insight into data abstraction architecture. Alan Crocker and Brian O'Hagan of NASA Johnson Space Center's Mission Operations Directorate provided insight into current mission operations and the requirements for data abstraction. This work is funded under a NASA Ames Research Center Small Business Innovation Research Grant. David Alfano of NASA Ames Research Center is the SBIR COTR.

REFERENCES

1. R. James Firby, "An Investigation into Reactive Planning in Complex Domains," in *Proceedings of the National Conference on Artificial Intelligence*, 1987.
2. Fitzgerald, W., R. J. Firby, A. Phillips, and J. Kairys, "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.
3. Goudarzi, Sara and K.C. Ting, "Top Level Modeling of the Crew Component of ALSS, in *Proceedings of the International Conference on Environmental Systems (ICES), Society of Automotive Engineers (SAE*), 1999.
4. Kortenkamp, D., and Bell, S., "Simulating Advanced Life Support Systems for Integrated Controls Research," in *Proceedings of the 33rd International Conference on Environmental Systems (ICES), Society of Automotive Engineers (SAE*), 2003.
5. Kortenkamp, D., R. Peter Bonasso and Debra Schreckenghost, "Managing Life Support Systems Using Procedures" in *Proceedings of the 37th International Conference on Environmental Systems (ICES), Society of Automotive Engineers (SAE)*, 2007.
6. Martin, C. E. and R. J. Firby, "Generating Natural Language Expectations from a Reactive Execution System," *Proceedings of the 13th Cognitive Society Conference*, 1991.
7. Spector, L. and J. Hendler, "Planning and Reacting across Supervenient Levels of Representation," *International Journal of Intelligent and Cooperative Information Systems*, Vol. 1, No. 3, 1992.

CONTACT

The authors may be contacted through email at scott@traclabs.com.