A Distributed Event Architecture for Space System Comps

Scott Bell
TRACLabs Inc.
1012 Hercules
Houston, Texas 77058
scott@traclabs.com

David Kortenkamp
TRACLabs Inc.
1012 Hercules
Houston, Texas 77058
korten@traclabs.com

Jack Zaientz
Soar Technology Inc.
3600 Green Court, Suite 600
Ann Arbor, Michigan 48105
jzaientz@soartech.com

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 [1] [2] [3]. NASA has created software processes called comps (short for “computations”) to manage these data abstractions. Comps are used in a variety of applications at NASA, from the execution of procedures to monitoring. Currently the comp creation process is manual, ad hoc, and intermingled with data displays. It is manual in the sense that comps are hand-coded computer programs for each data item. It is ad hoc in the sense that each comp is developed on its own with no representation of how it relates to the tasks being performed or to other comps. It is intermingled with the data displays in that comps are irreducible, difficult for other programs, like controllers, to access or understand, and often compiled into the data display. In this paper we present an architecture that allows engineers to design comps in an efficient, flexible, and canonical manner.

2. COMP ARCHITECTURE COMPONENTS
The architecture consists of several integrated components and representations. These include:

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

Data source The producer of data events. 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. This receiver may include high-level control systems, crew displays, logging or maintenance systems, or other abstraction architectures.

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

Sensor event abstraction language (SEAL) An XML grammar which defines the comps, the comp’s message handling operations, and the directed graph connecting the comps.

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.

2.1 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 [4]. 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.

2.2 Editor
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 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.

2.3 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 comps, connects the comps to each other, the sources, and the sinks[4]. When DARE is finished initializing, data sources are producing events from live data, comps are running on those generated events, and sinks are consuming the resultant events. 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 comps are accessible using a variety of protocols. DARE can be remotely debugged using the SEAL visual editing environment using JMX.

3. DEMONSTRATION

The demonstration we will show receives 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 [5]. Simulation progresses in hourly time increments, with each unit process producing and consuming various resources in designated stores. random failure and stochastic performance. BioSim has been successfully used and verified in many life support optimal design applications. 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.

We created an abstraction network to monitor five sensors that measured the carbon dioxide in the crew cabin. Nominaly, 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 1.

First, each carbon dioxide sensor is sampled to 1Hz. This means each Sampler comp is generating one event every 1 second. These Sampler events are all consumed by the Temporal Alignment comp. This comp 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 comp 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 comp. 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 1. figure 2 shows the sensor values and the event detected by DARE.

4. REFERENCES