

COST EFFECTIVE GROUND ARCHITECTURE: A LAYERED APPROACH TO AUTOMATED MISSION OPERATIONS

Roger Thompson
SciSys Ltd., UK

roger.thompson@scisys.co.uk

Julian Long
SciSys Ltd., UK

julian.long@scisys.co.uk

Ivan Dankiewicz
SciSys Ltd., UK

ivan.dankiewicz@scisys.co.uk

1. INTRODUCTION

Through a series of studies and prototype developments, SciSys is developing an approach to low-cost ground segments that support automated mission operations, based on an open, layered architecture, consisting of Data Access, Operations Language, Procedure and Schedule layers.

The Data Access layer is consistent with current standardisation activities within the OMG Space Domain Task Force and combines a publish/subscribe approach to data distribution with a remote action interface for an extensible set of core data items. The core data items include status parameters, control commands and alert notifications, but may also be extended for a given system to include observation orders, planning requests, orbit vectors, etc. The system model for this data access layer is defined in an XML schema.

The Operations Language layer provides a procedural extension to the declarative system model, allowing engineering staff to define operations (expressions, conditions, rules and scripts) that reference and invoke actions on the objects at the Data Access layer. ICOL is a Java implementation of this layer, developed by SciSys that can be embedded in both C++ and Java applications. The ICOL definition environment checks consistency of operations against the XML system model and generates executable Java that can be invoked by the ICOL execution environment. ICOL can be used directly for applications such as parameter derivation, command checking and synchronous test scripts.

Automation within spacecraft mission control systems is often restricted by the scope of scripting languages or external interfaces provided by the spacecraft control system system used. Using the ICOL operations language layer as a base, SciSys is currently developing an automated procedure execution system (APEX), a lightweight Java implementation of its UNiT graphical procedure automation system, that has been successfully deployed in a number of multi-satellite control centres. APEX procedures will themselves be expressed in XML, and support execution of procedures defined in ESA's PLUTO language.

2. COST EFFECTIVE GROUND ARCHITECTURE

Smallsats, individually or in constellations, are increasingly being considered as a cost effective means of implementing commercial or service-oriented Earth Observation missions. As the cost of the space segment falls, there is an expectation that there will be a commensurate fall in the cost of the associated ground segment and mission operations. In practice, however, the ground segments require all the same elements found in their larger institutional equivalents, plus the additional complexity of providing open access to the services provided and increased automation.

BNSC has partially funded a number of studies that has enabled SciSys and its partners to define a Cost Effective Ground Architecture (CEGA) for such missions.

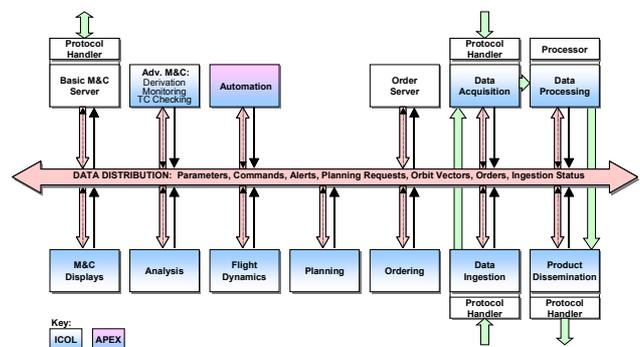


Figure 1: Cost Effective Ground Architecture

The CEGA architecture, illustrated in Figure 1, is an end-to-end system integrating User Ordering, Mission Planning and the Data Processing System via the same central backbone to provide a generic framework to control the information processing and product dissemination. Mission specific data processing algorithms, image display and product archive solutions may be added into this framework to provide the services required by the user.

The modular and layered approach is designed to allow rapid configuration and integration of mission specific systems from generic components; and common

implementation of interfaces, data access and data manipulation.

An overview of the technology elements central to the CEGA Architecture is shown in Figure 2 below.

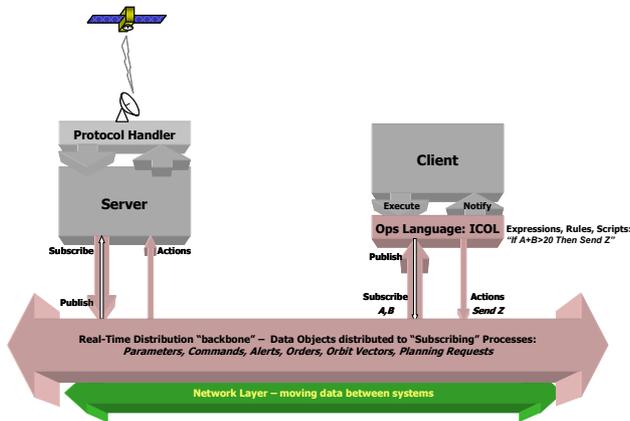


Figure 2: Schematic of Core CEGA Technologies

A central feature of the architecture is the real time data distribution layer that integrates all of the ground segment elements onto a common backbone. Generic data types representing the fundamental information shared between the ground segment applications are made available through this layer. This is consistent with the M&C Data Access service proposed by the OMG Space Domain Task Force, but extends the concepts to address order handling and data processing as well as monitoring and control. There are examples of existing systems which support such a service, implemented using CORBA of which SciSys has direct experience.

The CEGA architecture also insulates its core applications from any mission specific protocols used in communication with external systems, including the space segment. A Protocol Handler is used to interpret the communication protocols used and to extract the embedded data. For example, a protocol handler is used to handle the monitoring and control interface to the spacecraft. Commands are encoded before transmission to the spacecraft and parameter data is extracted from the spacecraft telemetry and made available to applications via the data distribution layer. The concept of Protocol Handlers is used throughout the CEGA architecture, wherever data crosses an external interface. Hence, the core ground segment can be truly 'generic' in it's handling of 'mission' data irrespective of the data formats and transport protocols used to get it to the ground segment.

CEGA allows the development of standard ground segment services and the tools to support them. These can then be mapped to external protocols. Classically, the space-ground interface and its protocols dictate ground segment

design. Protocol Handlers may themselves be generic where they support a standard protocol, such as CCSDS Packet TM/TC or a manufacturer's own proprietary protocol.

Protocol Handlers are used in the following interfaces:

- Space to ground links
- Ingestion of Ancillary Data
- Dissemination of Products.

The powerful combination of a standard interface to mission data coupled with the use of Protocol Handlers makes the CEGA architecture very flexible in the number of configurations the core system may be deployed in. The addition of functionality through extra applications may be introduced in line with the business plan and needs of the service. As the service establishes itself the ground segment may be extended to cope with additional demand or to maintain service timeliness. New dissemination technologies may be incorporated into the ground segment and offered as part of the service with minimal cost and impact through the simple addition of an extra Protocol Handler to convert the products to a different format or transport mechanism.

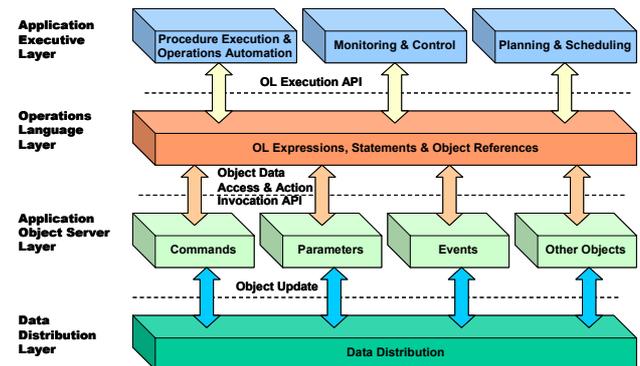


Figure 3: Layered Architecture

The layered approach to the CEGA architecture is illustrated in Figure 3. This shows client applications accessing core data items through an Operations Language layer that provides the means of combining and manipulating the data items, in terms of complex expressions, conditions and scripts that can be defined by engineering users rather than software developers. This layer also delivers the objective of an open, modular design by allowing any application conforming to the interface to be integrated into the ground segment, and available through the data distribution layer.

The Operations Language layer is bound through the Application Object Server layer to the underlying means of data distribution. Where the Data Distribution layer is itself standard, this integration only needs to be performed once

for each class of data item, and is then available for use by any application via the Operations Language layer. The abstraction of the Application Object Server layer allows legacy applications and alternate data distribution solutions to be integrated into mission specific system.

3. DATA DISTRIBUTION LAYER

The CEGA architecture described previously identifies a Data Distribution Backbone. It is intended that this should be a generic layer that will be applicable to a wide range of future Space Ground Systems.

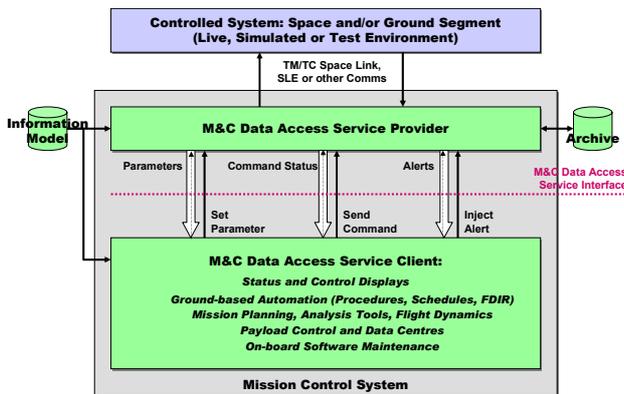


Figure 4: OMG M&C Data Access Service

The OMG Space DTF RFP2 addresses the development of a standard specification for a Space Domain Monitor and Control Data Access Service [see Figure 4]. The DD layer described here is based on the requirements listed in the RFP, but extends them to support additional Space Domain Types.

The Data Distribution (DD) layer comprises a set of Data Access Services providing access to instances (*data items*) of a set of generic *data types*. These Data Access Services provide the interface between client applications that are the consumers of data items and which may act upon them, and the service providers that maintain and publish those data items. For any given data type, there may be a number of distributed service providers, partitioned by the scope or “domain” of the data items they are responsible for. For example there could be multiple TM Parameter service providers, each servicing a single satellite domain. There may be many TM Parameter clients, corresponding to various applications: displays, automation, analysis, etc.

Each service provider is responsible for the instances (data items) of one or more classes (data types) that are made available to the various client applications via a Publish and Subscribe model.

The service provider is an integral part of the DD layer, however it can only provide a service by interfacing with a

system server which is an underlying server function of the ground segment. The system server may in turn obtain data via a protocol handler from an external system.

The following data types have currently been identified as part of the core CEGA architecture:

- **Parameters:** status information directly telemetered by the controlled system, or derived/maintained by the system server.
- **Commands:** any symbolic control directive, to the spacecraft, other controlled systems, or the system server itself.
- **Alerts:** raised asynchronously to report a significant event occurrence or anomaly.
- **Planning Requests:** an operational activity that requires scheduling..
- **Order:** a user request for a data product, which may require scheduling of spacecraft activities and contacts.
- **Ingestion Status:** the reception status of auxiliary data required for payload processing.
- **Orbit Vector:** a description of the satellite orbit trajectory.

The DD layer consists of service providers, which provides two types of access for each supported data type within the context of a specified *session*:

- **Data Delivery** – illustrated in Figure 4 by the block arrows. For each data type the client application initiates the transfer of data by subscribing to a specified subset of data items that are published by the service provider. In response, the Service Provider supplies the current status of each subscribed data item and thereafter it’s evolving status.
- **Actions** – illustrated by upward solid arrows. In each case, the client application can manipulate data items as follows
 - Value Assertion [e.g. set parameter]
 - Instance Creation/Destruction [send command]
 - Method Invocation [acknowledge alert]

The DD layer offers two standard APIs.

The *Data Distribution Interface* is a standard interface, which allows clients to subscribe for data items, and subsequently receive updates. The *Data Distribution Interface* also allows Clients to invoke actions and be informed of the action result. The use of the standard interface ensures that clients require no knowledge of the System Servers.

The *System Server Interface* defines a standard interface, which the DD layer itself uses to communicate with the System Servers. The use of this interface allows the DD layer to be reused with different underlying systems. It is likely that the DD layer will support more than one implementation of this interface. e.g. a direct interface may be developed which is linked in with the system server to allow the system server to efficiently update the DD layer but it may also be necessary to provide a networked interface to allow the system server to execute on a different platform to the DD layer.

The DD Layer supports multiple *sessions*. The term *session* is used with respect to the DD layer, to refer to a coherent data source, which may be represented by any of the following:

1. An Operational Satellite [or other controlled system]
2. A Satellite in Test Configuration
3. A Satellite Simulation
4. Dynamic Replay of Historical Data from archive. *Sessions* of this type do not support actions.

4. OPERATIONS LANGUAGE LAYER

The role of an Operations Language (OL) is to support the configuration and execution of control scripts required for the operation of a particular target application. The control scripts contain expressions and flow control statements which may reference application data as operands and call application actions during their execution. Re-use of previous operations languages has been restricted by factors such as non portability, application specificity, inadequate performance or a non intuitive syntax. These factors have combined to restrict standardisation in this area. The Integrated Common Operations Language (ICOL) attempts to address these issues.

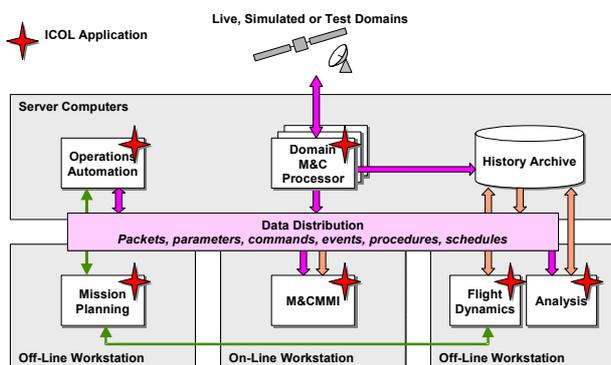


Figure 5 Potential applications of ICOL

ICOL is a platform independent, Java based environment, capable of providing common OL script and expression

execution services across multiple spacecraft control applications. Figure 5 shows the potential target applications which require an embedded operation and expression execution component including : TM/TC processing; Status Displays; Procedure Execution; Automated Actions; Mission Planning and Scheduling; and Test and Check-out Systems.

The ICOL environment supports the definition of the System Model objects such as parameters, commands and events. These objects encapsulate both the data and actions which are available to a particular application. Operations control scripts encoded in ICOL can access application objects via the intuitive and convenient reference scheme provided. Thus during ICOL script execution *Application Data* required for expression evaluation can be accessed such as spacecraft telemetry parameter values. Also during script execution *Application Actions* such as sending commands or raising system events can be initiated. The ICOL script syntax supports operation algorithm definition by providing a rich set of built in data types, control statements and mathematical functions.

ICOL Architecture

The ICOL development environment comprises of suite of component libraries and APIs, implemented in Java, which can be integrated into target applications written either in Java or C++. Integration to C++ uses a suite of wrapper objects around an interface implemented by the Java Native Interface (JNI). Figure 6 shows that ICOL components provide services for both the offline definition and on-line execution of ICOL scripts.

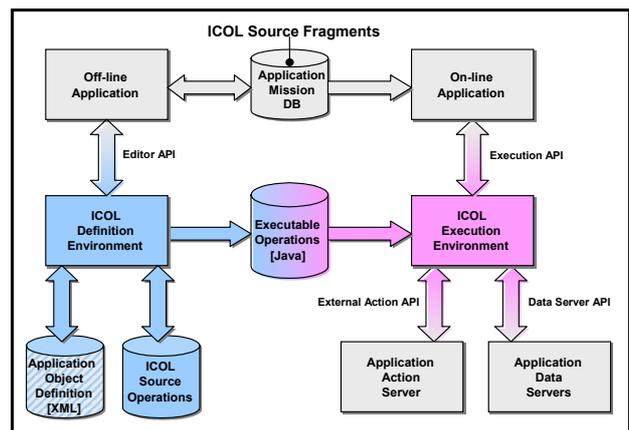


Figure 6 ICOL Component & API Architecture

The *ICOL Definition Environment* includes an Editor API which allows ICOL source code to be cross referenced and type checked against the *Application Objects* defined in the System Model before being compiled into executable Java byte code. The *ICOL Execution Environment* consists of

APIs which allow ICOL control script execution to be initiated and which support the referencing of *Application Objects* from expressions via the Data and Action servers.

ICOL Concepts

The ICOL scripting syntax is based on the portable Java language and which has the additional benefits of being an industry standard modern structured programming language supporting a rich set of in built types, functions and control statements. However, as it is intended to be written by spacecraft operations engineers who are assumed not to be experienced software engineers, ICOL supports only a simplified sub set of syntax constructs and functions available in Java.

Access to both data and actions encapsulated by *Application Objects* is supported by an intuitive and concise referencing scheme. References to *Application Object* data can be used as operands in expressions written in ICOL. On ICOL compilation the data references are replaced by calls to the Data server.

The ICOL environment supports its own XML based syntax for the definition of the System Model *Application Object* types and instances which can be scoped into *domains* and *sub domains* representing the decomposition of the application model into a sub system hierarchy. ICOL supports Object Orientated type definition including data & action encapsulation and object class inheritance. ICOL also supports the partitioning of *Application Object* types and instance definition into different files which increases the flexibility of configuration control.

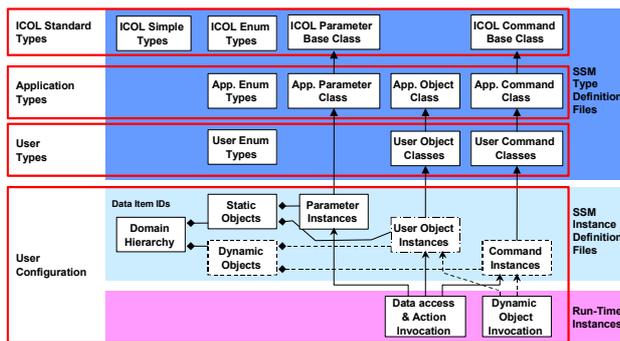


Figure 7 ICOL System Model definition files

Figure 7 shows the structure of the System Model definition files and the categories of *Application Object* types and instances that can be defined. The arrows between object classes show the direction of inheritance from a base class.

The *ICOL Standard Types* define all built in standard simple, date, time and enumerated types supported by ICOL. Also provided are ICOL Parameter and Command

base class definitions which can be instanced directly or form the base of an inheritance hierarchy.

Application Types can be defined which can use or inherit from any of the ICOL Standard types. These allow the definition of standard enumerated types and object classes that are configured to be used across a particular mission or application.

User Types which allow the definition of specialized enumerated types and object classes that are configured for a particular User or subsystem. User types can use or inherit from any of the ICOL Standard or Application types.

Performance

Performance is critical to the success of ICOL which is targeted to be embedded in applications which require a high rate of expression evaluation e.g. a telemetry processing kernel.

To benchmark the performance of ICOL and to determine the impact of using Java technology, three versions of an example application were produced for comparative performance evaluation. Table 1 shows that each version implements the same architecture but use either Java or C++ to implement the different layers.

Layer	Version	1 (hybrid)	2 (pure Java)	3 (pure C++)
Application layer	Executive	C++	Java	C++
ICOL Environment layer	Execution	Java	Java	C++
Application Object Server layer		C++	Java	C++

Table 1 Performance prototype characteristics

The tests were run on a 500MHz PC Pentium II with 256 Mbytes memory. The three sets of test data were generated each of which simulated the execution of 1000 distinct ICOL scripts. Figure 8 shows the performance of each version of the prototype application in executing the tests measured in average execution time of each ICOL script in micro seconds.

The results show that there is an approx. 50% performance penalty associated with pure Java ICOL execution compared to a pure C++ execution of the same ICOL script. This performance penalty is increased to approx. 300% if Java OL execution is integrated into a C++ application using the object wrappers implemented in JNI. This is due to the known inefficiencies of the JNI interface for function calling and data passing. Although significant, the measured performance overhead of using Java was anticipated and is within the expected range based on previous experience of Java implementation.

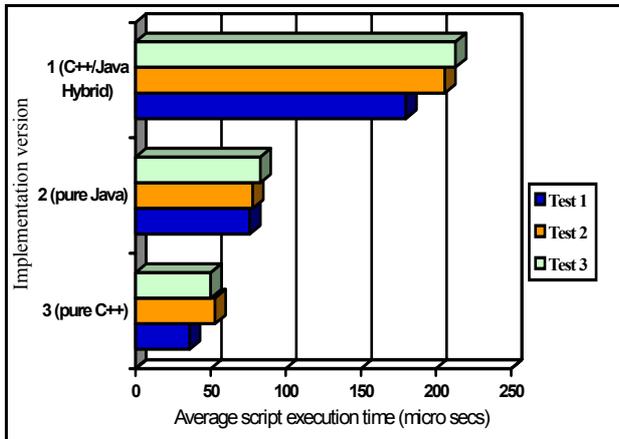


Figure 8 ICOL performance test results

From these tests we conclude that although the performance impact for using Java is significant; ICOL performance would be more than sufficient for most applications. Also the performance penalty for using Java is offset by its advantage of true platform independence for OL execution.

5. AUTOMATED PROCEDURE EXECUTION

The Automated Procedure Execution (APEX) product is a portable “lightweight” procedure execution environment that uses and demonstrates the data distribution and operations language layers outlined above. It facilitates the automation and testing of spacecraft operations when integrated into a monitoring and control system or a test and checkout application. APEX is targeted at low cost missions, including small satellites.

The APEX toolkit provides a generic environment designed to require the minimal integration and configuration effort for each new mission. To satisfy the requirements of the targeted low cost missions, emphasis is placed on a low deployment cost and a minimal resource footprint. To this end APEX is based on Java technology and open standards, without the use of expensive 3rd party COTS software. In particular Java based ICOL is used for expressions which control the execution flow of procedures.

The procedure model has been captured in an XML schema. The schema supports the users to define correctly structured procedures using XMLSpy which is a low cost commercial XML editor. Integration of procedure definition with ICOL allows the procedure control condition expressions to be checked against the System Model.

APEX builds on SciSys’ extensive experience and investment in providing automated monitoring & control systems, most recently through deployment of the UNiT toolkit for EUTELSAT, UK MOD and EUMETSAT. In particular, the proven conceptual model of UNiT automated

operations procedures has been re-used in the context of the APEX environment. This procedure model has been modified and re-implemented for lightweight execution of individual procedures, with clear separation between the execution engine itself and support for visualisation and display. This approach enables the distributed execution of procedures, resulting in a flexible and scalable solution for the deployment of automation. It is also consistent with a long-term goal of on-board deployment of procedures.

APEX Architecture

Figure 9 shows the APEX system architecture that comprises four major elements:

- **APEX Procedure Definition** element provides tools which supports the definition and checking of procedures.
- **APEX Procedure Execution** element provides a lightweight and portable environment for the execution of procedures.
- **APEX Procedure Display** element provides display components which allow the visualisation of single procedure status in detail and an overview status of all executing procedures.
- **APEX Procedure External Control** element provides an interface allowing procedures to be executed on request by an external source.

Figure 9 shows the decomposition of the elements into the main components and interfaces.

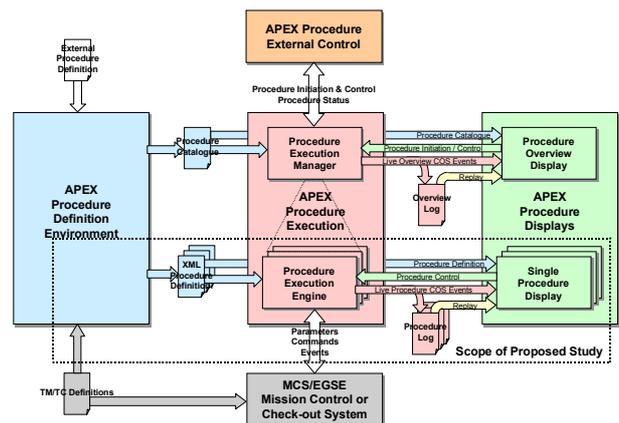


Figure 9: APEX System Architecture

Procedure Model

Procedures correspond to pre-defined operational activities that can either be scheduled as a ground-based *activity*, or manually initiated from a display client. *Procedures* may also call *Sub-Procedures* which permit decomposition into smaller and more maintainable units. This allows operational activities to be defined more flexibly such that they can be used in different operational contexts.

The definition of a procedure comprises its public interface (including arguments), local variables and a set of *Thread* definitions. Each procedure contains a primary thread and, optionally, a number of secondary threads shown in the Single Procedure Display Figure 10.

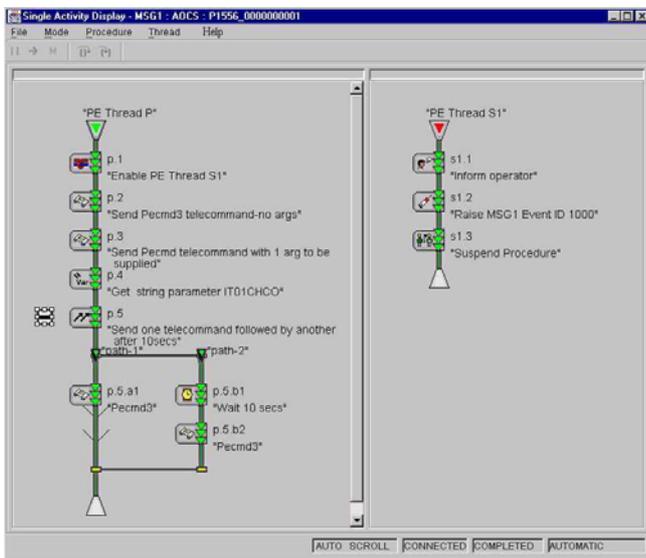


Figure 10 Single Procedure Display

Each *Thread* constitutes an independent flow of control through the procedure. The single *Primary Thread* corresponds to the main flow of control and is usually activated as soon as the procedure is started and continues to execute until the procedure terminates. *Secondary Threads* support monitoring or contingency functions that are performed in parallel with the *Primary Thread* within the context of the active procedure. *Threads* comprise an *Activation Condition* and a sequence of *Steps*. A *Thread* activates following the evaluation of its *Activation Condition* to true and then proceeds to execute the *Steps* in sequence. *Threads* can be enabled and disabled from other *Threads* to prevent execution if required by the procedure logic.

A *Step* comprises a *Trigger*, *Body*, *Confirmation* & *Recovery*. All of these components, with the exception of the *Body* are optional.

The *Body* of a *Step* determines what action it may perform which may be thread flow control, operator interaction, a *Sub-Procedure*, a *Sub-Thread* or a sequence of *Application Actions*.

Each *Step* is broken into three clear phases that of triggering, body execution and confirmation shown in Figure 11 :-

- **Trigger** - A *Step* may have an optional *Trigger* which controls the synchronisation and checking that conditions are correct to start execution. A *Trigger* consists of a Wait Condition and a Pre-Condition to check that the step is ready for execution.
- **Body** - Execution of the Step Body involves execution of the intended *Step* logic e.g. execution of an *Application Action* or flow control construct such as a loop or branch statement. An optional *Watch Condition* can be associated with the *Step Body* which is used to check that execution is proceeding normally such that, on failure, suitable *Recovery* contingency handling can be applied.
- **Confirmation** - Once the body is complete then an optional confirmation phase of the current step can be executed before moving on to execute the next Step of the sequence. A *Confirmation* consists of a Wait Condition and a Post-Condition to check that step execution has completed successfully.

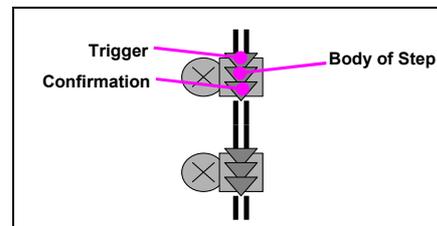


Figure 11 Phases of a Procedure Step

Recovery actions can be associated with the phases of *Step* execution. If a failure occurs during the execution sequence e.g. trigger fail, body fail or confirmation fail then appropriate *Recovery* actions are invoked.

In the procedure model described above all condition expressions used to control procedure execution are coded as ICOL expressions. The ability of ICOL to access application data, e.g. telemetry parameter values, in expression evaluation can then be harnessed to automatically control the flow of procedure step execution. Also low level atomic actions to be performed by a procedure such as parameter value assertion or spacecraft command initiation can be performed as ICOL actions.