Empowering the Check-Out User: Integrated Simulation

Gareth Patrick(1), Roger Patrick(1)

(1)TERMA
Schuttersveld 9, 2316 XG Leiden, The Netherlands
gak@terma.com, rmp@terma.com

INTRODUCTION

Check out systems and SCOE (Specific Check-Out Equipment) controllers provide numerous ways of visualising and monitoring a system under test. Standard features include synoptics and out-of-limits monitoring but it is also common for users to define other custom actions that, for example, process specific telemetry packets or launch automatic test sequences. The logic that drives this behaviour is encapsulated in the AIT working environment, either through database definitions or through scripts developed by the AIT engineer. However the ability to validate these features and tools is often compromised by the lack of a simple method to generate appropriate test data.

Simulators are of course in widespread use on space projects, and can be used to validate some function within the Central Check-out System (CCS) or SCOE controller. In reality the availability and complexity of external simulators means that they are not always a suitable choice: scenarios must be programmed into the simulator to achieve the required state (which can be just as demanding as working with a real system), and in some cases the ability to simulate the data stream that is required may simply not exist. Couple this with the fact that some CCS testing requirements may only come to light days before they are needed and it becomes clear that a simple, quick, built-in simulation function would be highly valuable to the AIT engineer. Knowledge we have gained through direct participation in spacecraft AIT activities has allowed us to identify and address this basic need of checkout system end-users.

In this paper we present a feature of the TERMA CCS5 & TSC products which integrates a simple yet powerful simulation function into the test sequence language itself. We show how it can be used to inject user-defined data patterns into the system based on packet structures that are driven by the mission database using just a simple high level syntax. This feature allows an end-user to very quickly construct the simulation scenario that they need in order to validate aspects of their AIT environment to a high level of confidence. But its uses are not just limited to those of self-validation. We also demonstrate how this simulation function can be harnessed by a SCOE controller in order to publish low level bus data to the rest of the system as a high level telemetry packets and parameters.

TSC and CCS5 Applications

The simulation functions described below are implemented in the context of the TERMA TSC and CCS5 software products. TSC is a lightweight package commonly deployed as a SCOE controller for instrument or sub-system level check-out activities. CCS5 is a multi-user CCS designed to coordinate larger-scale activities at system or spacecraft level. Both applications share the same code base and the same principle of operation. They each offer the same integrated test sequence language and provide the same views for telemetry parameter display and monitoring. TM and TC definitions are defined in MIB database ASCII format [1] (the same as that used by the SCOS-2000 system) and incoming telemetry packets are identified by the rules of the MIB tables.

The test sequence language used by TSC and CCS5 is termed uTOPE, and it is a re-implementation of the original TOPE test sequence language based on TCL as used in SCOS-2000 systems. As well as supporting the original TOPE commands, the development of uTOPE for modern platforms has provided the opportunity to add additional functions to the language including those related to telemetry packet simulation.

TM PACKET INJECTION

In a live setup external data is delivered to TSC or CCS5 via a deployment-specific EGSE carrier protocol, for which there are several industry standards. The specifics of the EGSE protocol are implemented via a plug-in module that effectively removes any encapsulation and delivers the TM or TC packet to the system. Typically the delivered TM packets will follow a CCSDS or PUS packet [2] format.

The uTOPE test sequence language provides a method for injecting data into the system as if it had been received externally. More precisely, binary data (representing a TM packet) can be self-injected at an entry point analogous to that of the plug-in module, meaning that TSC or CCS5 receive the data exactly as if it had been delivered to the system via the carrier protocol. Using this function a user can define any arbitrary or deliberate packet data that will be subsequently archived and processed by the system in exactly the same way as packets originating from an external provider. To retain traceability of injected data, injected packets are tagged at creation with a dedicated 'injection' property which is visible in the packet archives. Using this property it is always possible to distinguish between self-injected packets and those which have been received from external sources.

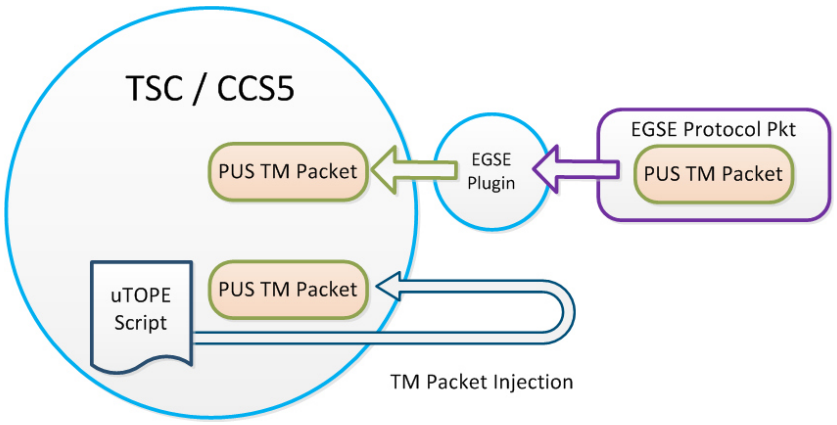

Fig. 1. Illustration of TM Packet Injection from a test script

Injection Command Syntax

Packet Injection is managed using the uTOPE command *processtmpacket*. In its simplest form (1) it specifies a binary string to inject. The system will then attempt to identify and process that binary data as a TM packet.

processtmpacket <raw> (1)

The uTOPE language itself is an extension to the generic TCL scripting language (which is also fully available in the working environment) so scripting file access is trivial, and there are also helper functions to convert strings from hexadecimal to binary. With this toolset it is little work to create a script that reads packet definitions from a file and injects them into the system. This makes it very easy to inject 'real' data that has been saved to a file (perhaps by another system), or to manage specific scenarios of packet data as separate files.

The basic syntax can be extended to specify a packet-ID corresponding to the PID_SPID field in the MIB database.

processtmpacket <raw> <spid> (2)

When used in this form the injected data given by <*raw*> is processed as if had been identified as the packet <*spid*>. The APID, PUS Type and Sub-Type associated with the <*spid*> in the MIB database are assigned to the packet and the packet cargo is processed according to the packet structure associated with <*spid*>. Essentially, this option skips the identification logic in the TM processing chain, meaning that the data given by <*raw*> does not need to specify a representative packet header; identification can be forced at run-time.

Extending the syntax again the <*fabHdr*> Boolean flag now specifies that the system should create the packet header dynamically and prefix it to the data given by <*raw*>.

processtmpacket <raw> <spid> <fabHdr> (3)

In this case the packet header will be created based on the MIB definition of packet <*spid*>, which includes the APID, (and Type and SubType fields for PUS packets). Packet Length, Source Sequence Count (SSC) and Cyclic Redundancy Check (CRC) checksum fields will also be filled dynamically. This format of the *processtmpacket* command is particularly effective for publishing low level bus data to the system as a TM packet where <*raw*> might consist of the datawords retrieved from a Front End interface. This is discussed later.

The syntax can be extended again with the <*fabBody*> flag, which instructs the system to dynamically generate the packet body as defined by the structure associated with <*spid*> in the MIB database. In this case <*raw*> would simply be an empty string.

processtmpacket <raw> <spid> <fabHdr> <fabBody> (4)

The system will create the packet body using the current values of the telemetry parameters and then prefix this body with the corresponding header as already described. So in effect it is possible to generate a fully populated and fully coherent packet (header, body and checksum) for any <*spid*> using just a single line of code and without having to explicitly define the fields within the packet.

In its most complete form the *processtmpacket* command allows the user to override the default packet body generation and specify his own values for parameters inside the packet itself.

processtmpacket <*raw*> <*spid*> <*fabHdr*> <*fabBody*> <*rawParams*> <*engParams*> (5)

Here, <*rawParams*> and <*engParams*> are standard TCL (associative) arrays which define the values of specific TM parameters using the parameter name (PCF_NAME) as the array index and the parameter value as the array value. Parameters can be specified in terms of raw or engineering values via the appropriate arrays. The beauty of this approach is that the user can refer to each parameter simply by its name in the arrays (as defined in the MIB) and need not be concerned with packet structure itself; parameters will be automatically placed in the correct locations in the packet body when the injected packet is created. This implementation is also designed to be flexible and does not require every parameter in the packet to be defined; any parameters that are not specified in <*rawParams*> or <*engParams*> are simply filled in using their current values in telemetry.

Should finer control be required additional options allow the user to override fields with their own values. The <*obt*> and <*ssc*> options allows the user to define specific values for OBT (On Board Time) and SSC that should be used in the packet header when the injected packet is created.

SELF VALIDATION

The feature set described above provides the AIT engineer with a simple yet effective method for simulating arbitrary telemetry streams via a test script. Packets can of course be injected when the system is 'offline' and unconnected to real SCOEs meaning that the AIT working environment can be self-validated completely independently from the wider ground system. Typical uses might include verification of:

- out-of limit detections
- synoptic behaviour
- automatic test sequence triggering
- post processing triggered by certain packet instances

The fact that the simulated packets are managed by test script also means that validation scenarios can be adapted quickly, or even dynamically. For example, the database can be interrogated via the test sequence language itself in order to know the out-of-limit values to simulate. Importantly, the packet generation function is database driven, meaning that should a packet structure or transfer function change it will automatically be taken into account; there is no impact on the test sequence itself. All of this drives towards self-validation possibilities that are both simple and low maintenance.

FURTHER APPLICATIONS

Self-validation is an obvious application, but the ability to inject user-defined TM packets can also be used to implement more sophisticated simulation scenarios within the CCS working environment.

Simulating TC Verification

A common obstacle when trying to validate test sequences in an 'Offline' configuration is that telecommands will fail, either because the system is disconnected from SCOEs, or because there is no system or simulator generating the appropriate responses to a telecommand. When a telecommand request is made from the CCS the TC has to transit various verification stages. Some of these relate to the ground system and are managed by the plugin and EGSE protocol; others relate to the flight segment and are derived from responses by the On-board software (usually consisting of PUS service-1 TM packets).

To facilitate offline validation the CCS5 and TSC environment provides a mechanism for sending telecommands in a modelled mode. When a TC request is made in modelled mode (either manually or via a test sequence) it is injected into the system as normal, but is never actually delivered to the plugin. Using the uTOPE *updatecommand* statement it is then possible to force the status of any of the TCs verification stages from a test script. By adopting this method a test sequence can mimic the verification triggers that would normally come from other parts of the ground segment, so the system behaves as if connected to a real receiving entity. Fig. 2 shows this schematically.

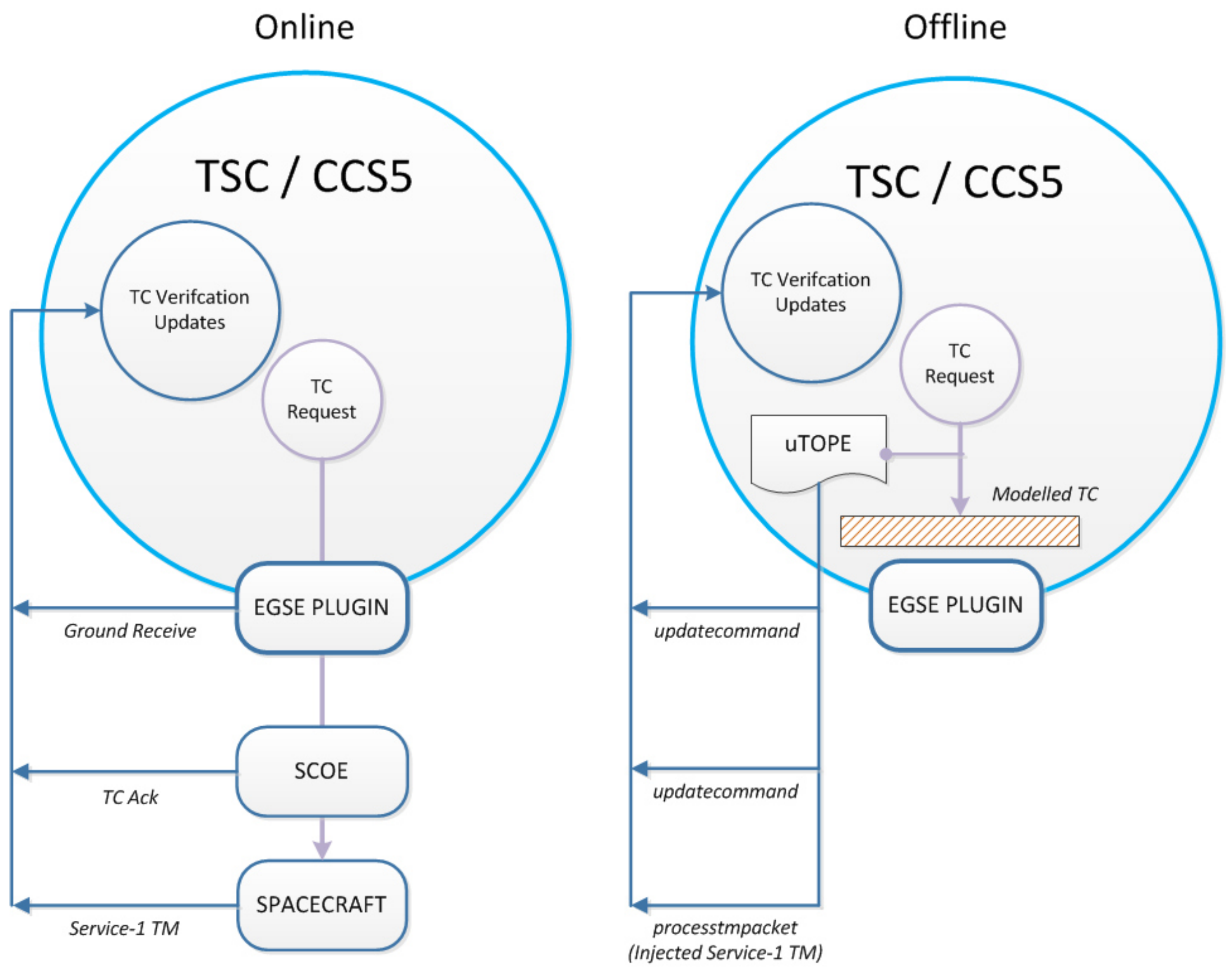

Fig. 2. Simulating telecommand responses in an offline configuration

Verification stages relating to the space segment can be self-simulated using the packet injection feature described previously. In the uTOPE test sequence language it is trivial to subscribe to telecommands and intercept them with a callback procedure whenever a TC request is made. This callback procedure can identify the sent TC, extract its header, and use it to compile the body of a PUS service-1 telemetry packet that will be injected into the system at a given time in the future. With a small amount of effort it is possible to simulate (script) a complete verification profile for any arbitrary telecommand.

The ability to self-simulate telecommand responses means that the AIT engineer is truly autonomous. Development cycles can be performed more quickly and test sequences can be validated to a higher level of confidence before being used on the specimen itself.

Publishing Low-Level Bus Data

At subsystem level a SCOE controller will often interface with the system under test using a low level hardware protocol. A typical example might be a Milbus-1553 controller where the unit under test is controlled and monitored via dataword messages on the bus. In such scenarios the system receives data in its native ‘raw’ format, but there is often a need to express this data in terms of parameters in order to monitor and exploit them in the AIT environment.

One way to achieve this is to define an ‘internal’ telemetry packet definition that maps (for example) the 32 datawords of a Milbus message onto a set of parameters with their own encodings and transfer functions. By using the *processtmpacket* in form (3) and specifying the Milbus message as the raw binary string the low-level data will be injected into the system as a TM packet and via the standard TM processing chain its content will be made available to the system in the form of TM parameters, as depicted in Fig. 3. The main work required to implement this is in the ‘internal’ TM packet and parameter definitions, which need to describe the information in the bus data. Thanks to the ability in CCS5 or TSC to combine multiple databases at run-time these definitions can be managed in a separate MIB database within the AIT environment.

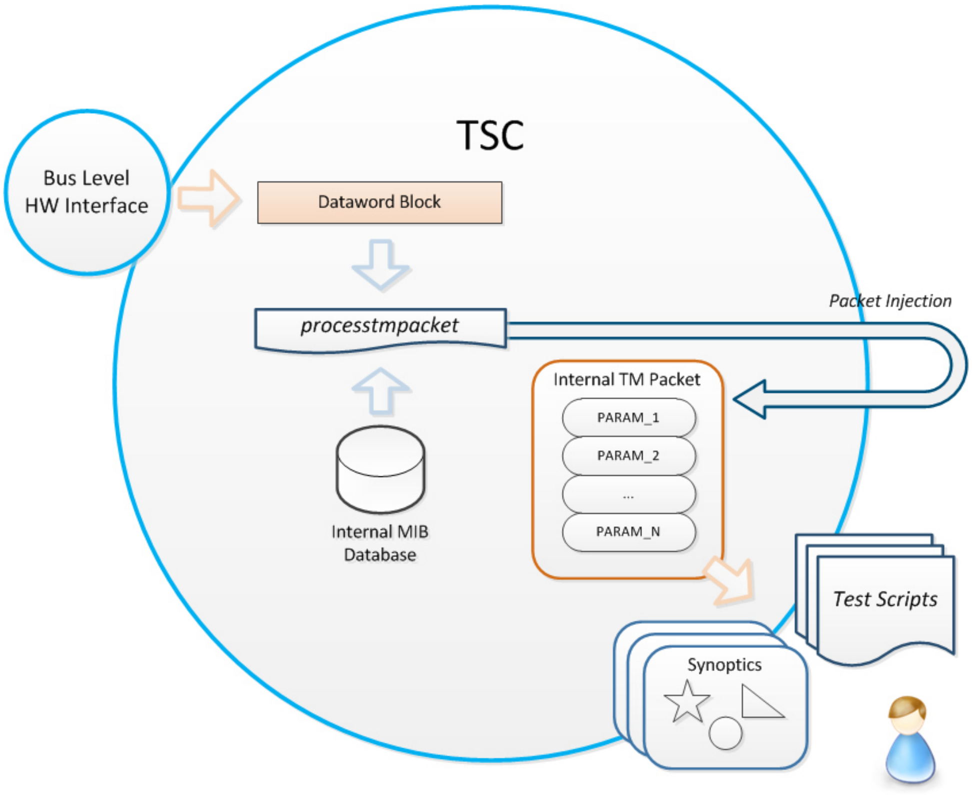

Fig. 3. Publishing low level bus data to the system as telemetry parameters using *processtmpacket*

Simulating a Spacecraft Interface

Very often the early phases of subsystem development will be performed in isolation using a dedicated instrument SCOE or EGSE to communicate with the system under test uniquely via bus-level data structures. At this stage a low-level interface with the hardware is necessary for testing and debugging, and validation test scripts are defined within this context.

However, once integrated within the spacecraft (or simply a larger system) the low-level databus is no longer directly addressable and the instrument must typically be commanded and monitored using high level TM and TC (e.g. PUS formatted packets). The transition from a low-level bus interface to a high-level packet interface can often be a barrier to re-use and dictates that two families of test/operating procedures need to be developed. However, by using the packet injection function the SCOE controller itself can actually implement the high level packet interface.

In the previous example we showed how raw milbus data could be published back to the system as telemetry parameters using an internal set of packet definitions. If we now imagine that these parameters need reformatting into a spacecraft PUS packet structure it is simply a matter of performing a second round of packet generation, this time referring to a <*spid*> that defines that PUS packet. A test script can easily instantiate a TCL array of <*rawParams*> based on the current values of the Milbus parameters, and use this to populate an injected PUS TM packet. In practice the TM packet would be generated using the *formattmpacket* command, which is analogous to *processtmpacket* except that it returns the binary string of the TM packet rather than injecting it. This binary string can then be handled in a test sequence and routed from the SCOE controller to the CCS. The result is that the CCS will see high level PUS TM coming from the instrument SCOE.

To complete the loop, in the telecommand direction it is relatively straightforward to run a handler script on the SCOE controller to process high level PUS TC from the CCS that are destined for the system under test. Contents of the PUS TCs can be translated into low level instructions on the databus that are sent via the local interface to the hardware.

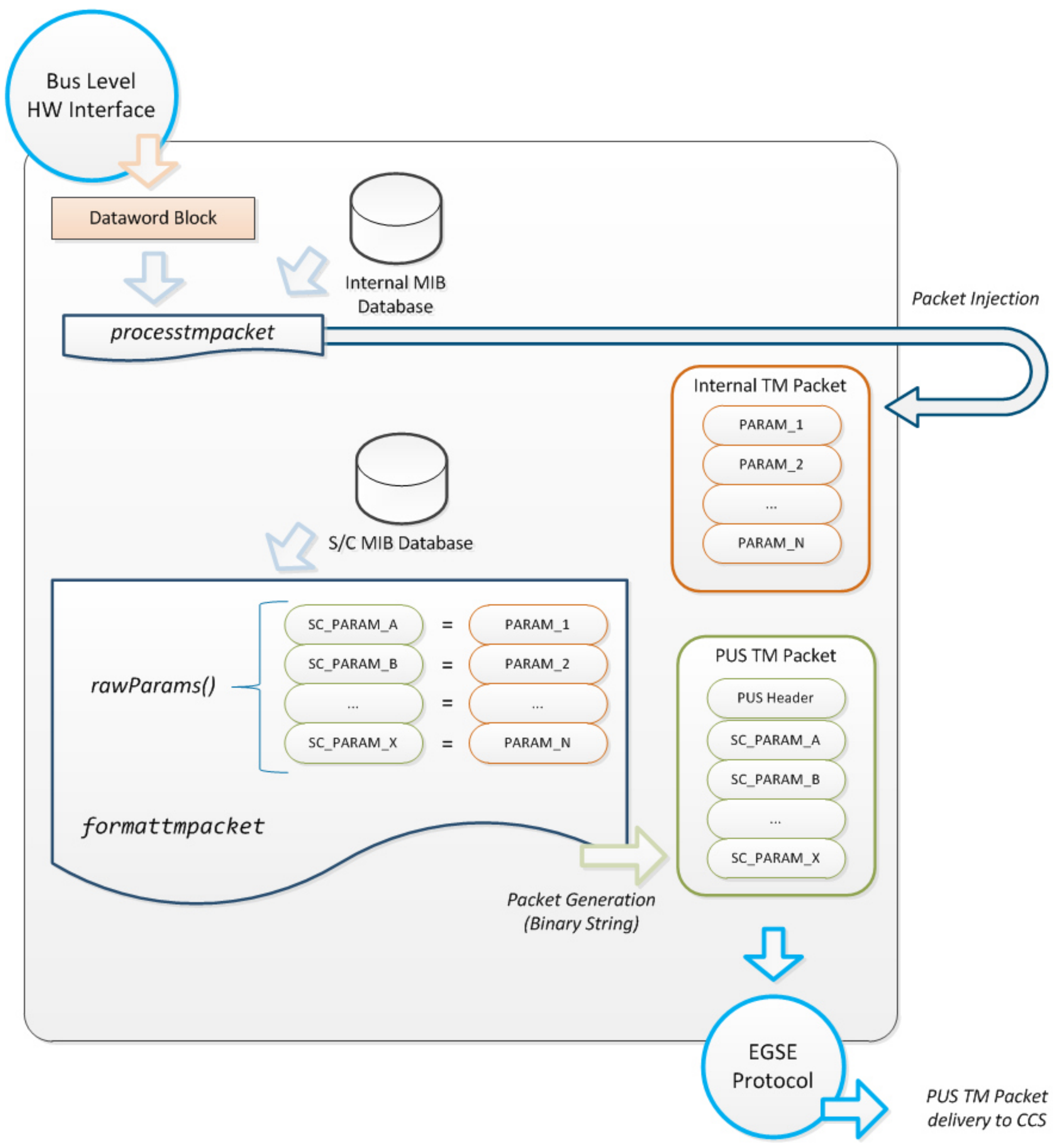

Fig. 4. Reformatting low level bus data and distributing it within a high level PUS packet structure

In essence the SCOE controller is simulating the role of the On-board Software, packaging low-level data structures into PUS packet formats and then transmitting them to other entities. This approach reduces the need for dedicated simulators and maximises the potential for procedure re-use during the AIT campaign, even as far as allowing use of flight operations procedures in an AIT environment.

LIMITATIONS

The *processtmpacket* and *formattmpacket* functions do allow variable packets to be injected, but only when specified as a raw binary string. It is not currently possible to define the content of a variable packet using the <*rawParams*> and <*engParams*> options. (However, packets with supercommuated parameters can be defined in this way).

SUMMARY

The simulation functions presented above allow the user of a CCS or SCOE controller to inject user-defined telemetry packets into the system via the test sequence scripting language itself. We have shown how these functions can be used for self-validation, for simulation, and for facilitating the manipulation of low level bus data.

Using TM packet injection and TC modelling functions the AIT engineer can work autonomously and is neither dependent on the availability of an external simulator, nor restricted by its capabilities. The simple syntax of the simulation commands and the flexibility of the TCL scripting language mean that the construction of test scenarios for self-validation is quick and intuitive to implement. Scenarios can be tailored to meet the specific needs of the AIT engineer and used to validate the AIT environment to a high level of confidence.

By using the packet generation function in association with low level bus data it is possible to publish telemetry parameters to the system quickly and efficiently via the standard TM processing chain. Using the same technique, low level bus parameters can be packaged into high level PUS packet formats with minimum effort, meaning that the SCOE controller application can simulate the function of on-board software. The details of parameter positions and encoding are all confined to the TM/TC database and need not be managed at all at test sequence level. Packet simulation is database driven meaning that simulation scenarios will automatically take into account any changes to packet definitions.

FURTHER DEVELOPMENT

It is becoming an increasingly common requirement to work with TM and TC data at frame level as well as packet level. A natural extension of the simulation functions would be to develop similar tools for the generation of TM and TC frames for which there are currently no lightweight simulators.

CONCLUSIONS

The main benefits of the packet injection functions in the CCS5 and TSC products can be summarised as:

- Autonomy: the end-user is self-sufficient and not dependent on external simulators.
- Quality of Validation: the end-user can inject any TM packet content in order to validate an aspect of the AIT environment (synoptics, limit checking, automatic test sequence triggering etc.) Simulation scenarios are scripted, automated and repeatable.
- Self-Simulation: Responses to telecommands can be easily and accurately simulated meaning that test sequences can be validated ‘offline’ to a high degree of confidence.
- Interface-Simulation: Low Level bus data can be published to the system efficiently as telemetry parameters. SCOE controllers can package bus data into TM packets to expose a high level packet interface to the rest of the system. Checkout activities can use high level operational test sequences in conjunction with instrument level SCOEs.
- Low Maintenance: Because the packet generation algorithms are database driven changes to packet structures are automatically taken into account with no impact in the test scripts themselves.

Together these improvements to the end-user workflow bring consequent gains in quality and productivity, and improve the overall AIT efficiency during preparation activities.

REFERENCES

[1] SCOS-2000 Team, “SCOS-2000 Database Import ICD”, EGOS-MCS-S2K-ICD-0001, version 6.9, *ESA/OPS-GIC*, 6th July 2010.

[2] European Cooperation for Space Standardization, “Ground systems and operations — Telemetry and telecommand packet utilization”, ECSS-E-70-41A, *ESA-ESTEC Requirements and Standards Division*, January 2003.