Advanced Telecommand Verification

Andrew Armitage[(1)], Vidjaikumar Kalicharan[(1)]

[(1)]Terma B.V.
Schuttersveld 9, NL-2316XG, Leiden, The Netherlands
Email: aba@terma.com , vik@terma.com

INTRODUCTION

Telecommand verification means checking whether a command sent to a spacecraft was successful or not. To do this, we could send a command, and wait, while listening to telemetry signals. If telemetry shows the satellite is still alive, with all systems nominal, then all is well. We can at least assume the command did no harm, but we would probably prefer to see some confirmation that specifically our command had a specific effect.

Current systems achieve this reasonably well, and this paper will briefly outline how they do it. There can be a number of practical difficulties, for example our satellite may not always be visible from a ground station. Time-tagged telecommands are a common solution. Less common are cyclic commands, repeatedly emitted on board under certain conditions, for example at a certain orbit position.

Some current spacecraft and ones in development have software applications that send commands to one another, spontaneously or from stored files. They have services such as on board monitoring, event-action service, on board command sequences, on board control procedures and macro command processors. Some of these were long ago specified in the Packet Utilisation Standard (PUS) but never implemented until now. Others are manufacturer- or mission- specific. The general trend is for a higher degree of autonomy or automated control, a trend which goes hand-in-hand with higher performance on board computers and consequently more ambitious mission goals.

Unfortunately, ground control systems have scarcely kept pace with advances on board. This poses a problem because limitations of the ground control system should not limit the ambition of the satellite or its mission.

This paper explores different aspects of the problem domain, and describes an approach Terma has adopted in its in-house developed monitoring & control systems.

CURRENT STATE OF THE ART

In this section we consider ESA SCOS2000 capabilities in command verification. Other systems obviously have different implementations, but SCOS2000 is representative enough.

SCOS2000 TC verification is flexible, being defined in the MIB, the spacecraft characteristic database. The database tables define the verification criteria and the time windows in a soft way, whereas some other systems sometimes assume that all commands to the same spacecraft have the same verification criteria. SCOS2000 also uses a number of settings and environment variables to define additional global variables such as space link propagation delay and the amount of time jitter to permit in addition to the time windows permitted in the verification timers.

The first, most general method is to verify by TM parameter expected value. If a command is sent, we expect a TM parameter to achieve a specific value within a certain time. This is measured by defining a timeout window to open, and then observing the first parameter sample after the window opens. This sample is then required to be the expected value. This method has the advantage of being completely general. All that we need is to measure the value of a TM parameter. There are no special packets to define. This approach would work even with a non-PUS spacecraft. A disadvantage of this approach is that we have no explicit reference to the original command. If the TM parameter changes as expected, can we be completely certain that it changed because of our command, and for no other reason?

The second method uses an explicit message to indicate the progress of the command, when PUS is applicable, these are packets of Service type 1 (usually called TM1,x). These are special report packets returned from the spacecraft to indicate that the command has reached a specific stage successfully or not. The report packet should be received within a defined time range. The advantage of this approach is that it gives a (practically) unambiguous confirmation about one specific command progress. It applies to one command and no other. We say “practically” because the Source Sequence Counter (SSC) in the packets cycles round to zero. If this has happened during the TC verification period, then it could result in an ambiguous report. This problem occurred in one major science mission where the SSC range was reduced, and yet a large number of commands to the on board computer had to be stored in the mission time line. Another problem with this approach is that downlink bandwidth has to be allocated to the short report packets. The packets have

to be defined according to a protocol, in this case PUS, so it will not work for other types of satellite which do not use PUS. In general, of course other protocols can use a similar method to return confirmation messages.

Finally, SCOS2000 permits status consistency checks (SCC), which in other systems is sometimes called "change on command only" (COCOMO). As implied by the description, this approach treats a change to a parameter as an error *unless* a specific command has been sent. It means a parameter should *only* change when a command has been sent that would change it. As with the other verification approaches, we can define a time window after the command was transmitted: the parameter is permitted to change within that time window. After the window closes, the value may not change any more unless or until the relevant command is sent again.

SCOS2000 has different states to accommodate some of the more tricky scenarios that can happen. For example a command is sent which expects one value, following this, while the first command is still pending verification, another command is sent expecting a different value. In this case the first command TC verification has been "superseded", and SCOS2000 has a specific status for this case.

CURRENT LIMITATIONS

A minor limitation of the SCOS2000 model is that it allows only one type of verification on each stage. This is only potentially important in one case, where it might be useful to do both types of verification on command completion. Even if acknowledged by a TM(1,7) or TM(1,8) to indicate successful completion or not, we might reasonably expect to check a TM parameter as well. The data model of the MIB does not allow this. In several missions, this has been worked around by "borrowing" the last progress stage for this purpose. This permits commands to be simultaneously checked by both methods. The only disadvantage of this workaround is that it is using a stage for an unintended purpose, which might be confusing to the unfamiliar user. It might have been preferable to define an additional finalisation stage, so that the TM parameter-based conditions could be placed on this final stage.

The main limitations of the SCOS2000 model are due to the verifier model being encapsulated in the C++ code of the application, and not being easily accessible for mission customisation. The design of the verifier was initially supposed to permit an inheritance-based mission customisation, but in reality this seems not to have been used, instead being patched or different configurable options added. The result is quite complicated and difficult to maintain, although this is not the fault of the original designers, it is merely the result of very long maintenance.

The budget needed to customise the verifier for a given application is relatively high, because it has to be implemented in C++ and CORBA, and consequently any mistakes have to be corrected by recompilation and reinstallation.

Most current ESA missions define the PUS service type 11, on board TC scheduling, or MTL (Mission Time Line). SCOS2000 is quite smart in its approach to basic type 11 commands; if a command is to be executed at a certain time, then the verifier correspondingly modifies its timeouts to occur at the time specified in the MTL loading command. Unfortunately most ESA missions also now permit modifying the timeline in the spacecraft, for example by deleting certain groups of commands, or by changing the execution times of other commands, activating or deactivating commands by sub-schedule ID. In the ground system we would like the telecommand verification to follow suit correspondingly. Unfortunately this is complex to implement, and invariably mission specific; we are not aware of a single ESA mission that has followed the PUS standard exactly, and most missions we have encountered are significantly different in this respect. Unfortunately this leaves little scope for reuse from one mission to another, and leads to a rather high cost to adapt to a specific mission.

TERMA CCS5 GENERAL APPROACH

For economic reasons, a software product needs to be maintained as far as possible in a small number of variants. The system can be made highly configurable, but we cannot contemplate having a configuration control source tree or branch dedicated for every mission. The burden of testing and maintaining each branch would vastly increase costs.

For a checkout product, we also have to face the issue that during the satellite development and integration, bugs may be discovered in the on board software or in the mission specific protocol definition (the "Space Ground ICD"). In these matters, the on board system is paramount, and so the checkout system is often required to adapt to these changes. To go through a complete C++ bug-fixing, build, testing and re-delivery during the AIT campaign is slow, disruptive, unreliable and expensive, whereas typical small modifications could easily be done directly on site by the end user. A checkout system has a specific need for flexibility.

For this reason we decided to implement most of the features of the SCOS2000 verifier in C++, but where features were known to be different in each mission, allow them to be scripted in the test procedure language (TOPE). TOPE was a TCL-based language previously used in SCOS2000-based CCS, and then re-implemented for CCS5.

TERMA CCS5 IMPLEMENTATION

The system has a telecommand "cache" which is, in essence, a list of all recent commands undergoing verification or awaiting completion. The cache is visible from the user interface directly. It is configured with a certain size, above which completed commands can be discarded. If fewer TC have been sent, commands are kept, which provides a recent command history. After the cache size has been reached, the oldest commands will be discarded so long as they are completed.

The TC cache is allowed to grow, to hold all commands that are not yet complete. The user will be aware of the mission parameters, and will be able to estimate the required cache size in advance. The required cache size depends on the mission commanding rate, and the verification time-out, which is usually derived from the on board performance, and the communication round-trip delay. Obviously if there is a large verification time out, and we send commands at a high rate, then we will need a larger cache. Consider for example a 30s verification time out, a 10s propagation delay, and an intention to send 100 commands a second continuously. In this case we should plan for a TC cache size of:

$$(30 + 10) \ . 100 = 4000 \tag{1}$$

Notice that, even with a high commanding rate and a slow link, the cache size does not need to be especially big. If the cache size is forced to grow, then the system emits a warning, but keeps all the outstanding commands. To keep the original command and all its properties will consume some kilobytes per command depending on its actual size. The memory consumed by the entire cache will therefore be in the scale of a few megabytes, or in the worst case tens of megabytes.

The essential feature in CCS5 is the ability to access and modify the TC cache directly from TOPE. We can query the cache, access the properties of any TC, and update these properties to affect the embedded TC verifier behaviour. Note that a full CCS is a distributed system, consisting of a server and attached workstations. The system overall TC verifier operates on the server. For performance, any mission-specific scripts that will access the TC cache should preferably run on the CCS server.

Commands in the TC cache are identified by their command ID. This ID is allocated when the command is sent; it is a unique number across the complete system. The commands from different workstations are made unique by including the workstation ID in the upper 4 bits of the command ID.

TELECOMMAND PROPERTIES

All commands in the cache have properties, which represent the attributes of the command. The most fundamental property is "id" which is the command ID. There is also "raw" which means the raw data of the packet. Other properties are for example "apid", "generationTime", "executionTime", "mapid", or "bypass". Very often these properties are derived from the command definition in the MIB, or are specified when the TC is sent.

Properties are saved in the archive binary data as part of the archive header. A subset of these properties are indexed, so they can be retrieved using SQL queries. These properties are used by the system user interfaces, as well as being accessible to the user via the TOPE language. As a matter of fact, CCS5 saves all properties of all commands at every verification stage, so this means the CCS5 telecommand archive is quite greedy of disk space.

In TOPE we access properties of a TC using the "tcprop" command:

% tcprop <id> <property> ?value?

Clearly we need to know the id, and the property name we are going to query or modify. The optional value parameter indicates a new value for the property. If not specified, the command returns the current value for the property.

SCHEDULED TELECOMMANDS

The scheduling "mission time line" approach outwardly resembles SCOS2000. We can emit a command to execute at a certain time, for example:

% tcsend ZP_00003 executiontime 12345678.0 subscheduleid 4 checks {SPTV DPTV CEV}

As a result the system generates a "carrier" command from a configurable template. The TC verifier is aware that it should postpone verification until the specified time. Importantly, in the command cache, properties are visible, "timeTagged" is true, and "executionTime" and "subscheduleId" match the values just given. These properties are used by the user interface as well as the TC verifier. As in SCOS2000, these features are largely programmed in C++.

Where our implementation extends beyond SCOS2000 is in the possibility to change the time tag verification depending on other commands that may be sent. The solution is that the needed mission customisations will be programmed in the TOPE scripting language, and hence can be developed either by Terma or by the end user. Being scripts, minor corrections can be directly performed on site, and tested directly with the on board software during testing.

Initially we need to be able to determine when a command has been sent that might change the mission time line. TOPE provides the ability to subscribe to commands as they have been sent, and to filter the subscription to only those which would affect the mission time line. To do this we can filter by property values:

% tcsubscribe referby mtlTC {timeTagged true} {subscheduleId 5}
% tcsubscribe referby changeMtlTc {type 11}

The first example detects commands that will be *contents* of the mission time line, filtered by a specific subschedule ID, while the second example detects commands that *modify* the mission time line. For the purposes of this paper, the second example is perhaps more interesting, because we can parse these commands, or their properties, to work out how to change the ground-based mission time line model to match the on board time line.

The system provides a number of TOPE commands to manipulate the ground time line model:

Table 1 Mission Time Line Modification Commands

::utope::mtldelete	Delete filtered MTL commands from the cache
::utope::mtldisablerelease	Disable release of filtered MTL commands in the cache
::utope::mtlenablerelease	Enable release of filtered MTL commands in the cache
::utope::mtlreset	Deletes all MTL commands from the cache
::utope::timeshift	Shifts filtered MTL commands in the cache The TC verifier observes the modified execution times

CYCLIC TELECOMMANDS & MORE

Cyclic commands are sent once to the satellite but can execute multiple times. The classic use case is telecommands that execute repeatedly each time the satellite comes to a specific orbit position. The command is emitted on board based on some telemetry provided by the AOCS system, for example a GPS parameter. This has been implemented in SCOS2000 for a couple of missions, however the approach has been programmed in C++ specific to given missions, and are barely reusable. In CCS5 we have tried to implement a more general-purpose approach, using TOPE to model the conditions when command verification windows should be (re)opened.

The core building blocks, implemented in C++ consist of the *cyclicRepeats* property and the *updatecommand* TOPE function. Essentially all commands can be considered repeated if their *cyclicRepeats* property is more than 1. It can be a definite number (e.g. 3 or 4), or effectively infinite (-1). Each time a command repeats, this property is decremented by 1. The command is considered complete, and can be deleted from the TC cache, only when the property reaches 0. In principle, a normal command could be considered to have *cyclicRepeats* initially 1, which reaches 0 immediately after the first completion, whereupon it can be forgotten. Using negative values for "infinite" repetitions may seem tricky but the absolute value is also useful because it indicates how many times the command actually repeated.

The other feature useful for cyclic telecommands (and other use cases) is the capability to update the TC verifier from TOPE using *updatecommand*. If we know the ID of a command we can force closure of a TC verifier stage, or force it to reopen, in effect restarting the verification time window. The syntax is

% updatecommand <id> <stage> <state>

To "recycle" a command we have to set the stage state first to IDLE, this resets the model. Then, at the time we want the verification to begin again (the time window will open), we set the stage state to PENDING. After this, the TC verifier will anticipate the same verification conditions to be met, using the same time window.

A benefit of this approach is that it does not depend on the specific way cyclic commands are formatted in the current mission, nor does it depend on any hard-coded TM parameters. The TC verifier does not have to be deeply changed for a given mission. Admittedly the TOPE sequence needs to be written and this needs to take into account various subtleties, but this is far easier to implement and modify than the approach it replaces.

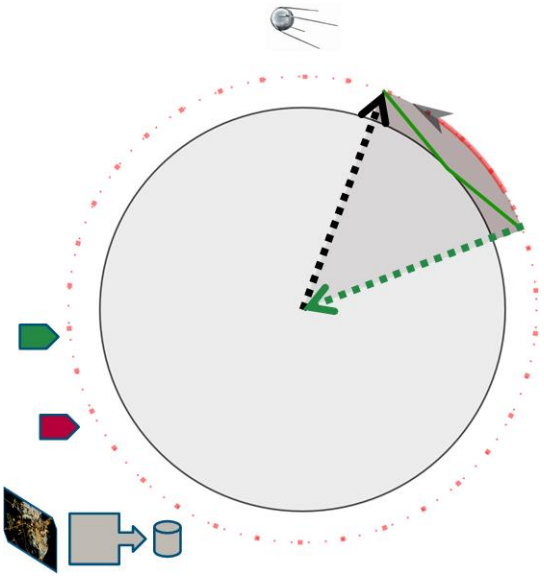

Figure 1 Orbit Position Scheduling

If we extrapolate this approach, we can see that there may be other general cases where commands are emitted on board. In general, if we have a way to model the conditions on the ground that would cause a command to be emitted on board, and we know its ID, then could (re)activate its TC verification. Setting aside the potential difficulties of getting a very accurate simulation, we can use a model to predict how the spacecraft has reacted to conditions on board; if the response includes emitting a command that was previously sent, whose identity we know in the TC cache, then we can also verify the commands emitted on board. Ultimately, this could potentially include simple cases like TC sequence files, but also on-board control procedures (OBCP) or dynamic cases like Event-Action service.

MODELLING TELECOMMANDS

TC modelling is a simple concept that can be applied in a number of different scenarios for different applications. In general it allows the CCS to track a command that was sent by another entity. This feature is not only useful for exotic future applications like on-board autonomy or formation flying, but can be used for some classic ground system or testing cases, such as the following:

(1) System Validation Tests (SVT) – here the MCS is used to command the satellite on the ground, while the CCS listens in. If the CCS only sees the commands from the MCS as binary strings, then it cannot verify the commands. However if the CCS has sufficient information from the MCS to model the commands, then it can set up its own verification conditions as though they were sent from the CCS and thereby verify them. It should be eminently feasible to pass sufficient information from the MCS to the CCS, obviously if both are CCS5, then this is trivial.
(2) Hot Redundancy – this can be applicable for critical test cases such as thermal vacuum tests, or for operations. Hot redundancy allows a second CCS server to take over if the main one crashes. A proper implementation allows the backup CCS server to show the history of commands sent by the primary one. If the backup server has to take over from the primary, it needs to be able to take over command verification, including time tag and cyclic commands as well as direct commands that were not yet completed.
(3) Command tracking within the EGSE. Different missions have use cases where an EGSE entity or SCOE is responsible for commanding the satellite or payload, while the CCS sees only the binary data. An example is the file transfer protocol CDFP implemented in a TMTC front end. If the CCS can reconstruct or receives information about which commands were sent, then the CCS operator can make more sense out of the commands sent by other elements of the EGSE.

Command modelling implementation is extremely simple. A modelled command has the property *isModelled* set to true. These are treated exactly like "real" commands, including verification and archiving, right up to the point of transmission. At the point of transmission the command is discarded. At TOPE level we can indicate that a command is modelled instead of "real" by using the *model* argument to *tcsend*, or by the equivalent *tcmodel* command.

MODELLING SPACECRAFT BEHAVIOUR

In the simple case of (for example) orbit-position based command scheduling, we do not need a very sophisticated model to know when a command would be emitted. In this case it is feasible to use a TOPE script to follow the orbit position reported in the telemetry, and to activate commands at the appropriate time. In reality we are "modelling" in a

very simple way the on-board software decision to emit a command when telemetry from the AOCS reaches the defined value, just as a time-based scheduler would with the on-board time.

Beyond this we have to consider what kind of simulation model might be needed for different cases. In some cases, TOPE or its equivalent may be sufficient, but in other cases, we might be better off using an external tool.

Table 2 Spacecraft Behaviour Modelling

Command Sequencing	This could be modelled with TOPE, because we only need to know a sequence of commands stored in a specific file. We need to know when the sequence starts, and the sequence contents.
On board Control Procedures	Debateable, different solutions are possible: (1) Use generic representation of the OBCP, e.g. in XML. Telemetry from the satellite could indicate the OBCP step reached, which allows to deduce which commands will be emitted next and thereby to model them (2) Pass telemetry to an external OBCP management tool like MOIS or APEX, and provide the external tool with an API to model the commands emitted by the OBCP as it executes. (3) Use an operational simulator, also with an API
Event-Action Service	Debateable, different solutions are possible: (1) To model the on-board response to a TM(5,x) is easily achievable in TOPE, based on subscribing to the event packets from the satellite. If there is a simple mapping to the emitted command, then TOPE can model it. (2) If there is a more dynamic decision executed on board, we may need to use a more sophisticated simulation, for example an operational simulator, or an emulator running the real on board software. This tool would need an API to insert information about the modelled commands. The modelling may be time-sensitive; and so the API needs to be efficient.

To integrate the CCS with external tools, capable of the most sophisticated command modelling, clearly the first step would be to provide an efficient external API for modelling the transmission of telecommands

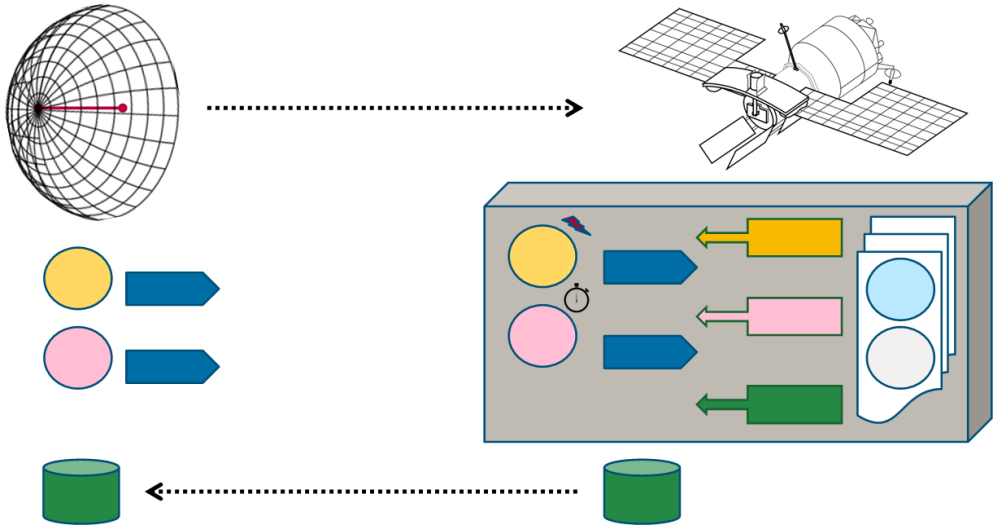

Figure 2 On Board Application Monitoring

NOTES OF CAUTION

This paper has considered some rather complicated and obscure scenarios. It is probably worth thinking twice whether the solutions might be over-engineered or too complex. A difficulty could be over-reliance on an imperfect ground model of the satellite behaviour. If the model timing is different from the real satellite, then this may cause the ground system to set up an incorrect model of commands emitted on board, which might lead to the ground system raising false warnings about non-acknowledgement of wrongly modelled commands.

If we arrive at a situation where the ground system is raising false command verification timeouts we have to consider what the operator will do next. The danger in sophistication is that when a very complex scenario goes even slightly

wrong, a human operator might have difficulty recognising whether the satellite is truly misbehaving, or whether it is only a problem that the ground model is imperfect.

We also need to be realistic about timing and synchronisation. There can be very high jitter in the communication round-trip time on a real ground network and radio link, while the link to our software simulator may be only a couple of milliseconds or less. If we are using a local ground model for comparison with the satellite in orbit, it is unlikely that the communication time to the model will very accurately model the same delay to the real satellite. It would be grossly optimistic to set very tight (millisecond level) command verification timeout constraints, because there will be high variation in the communication links. If this approach will be used, we would not recommend defining very strict timing constraints.

To what extent is it a blocking problem? Should we conclude, for example, that it is not worth modelling commands emitted from an OBCP?

In the authors' opinion, some difference between real satellite and the ground model is inevitable, but this may not be a very serious issue. We would contend that it is still worthwhile to model the commands emitted on board, even if there are a very limited number of false warnings. We could for example easily use colour in the MMI to indicate when a command was modelled; if we later see that the modelled command did not succeed, the operator can has a visual cue to remind him that there are two possible errors: either the satellite itself, or the model of the satellite.

On the other hand it is vital to avoid compound accumulation of differences with the on board model. If this happens then we might find that all modelled commands are verified too early, too late or not at all. To give a practical example, a TC sequence file could be defined with a fixed inter-command delay. We might attempt to model the inter-command delay by a timer in the ground system. The problem would be that any difference between the on board and ground timers (drift) would accumulate between each timer event. The error from each repetition would be added to the sum of all the errors in the previous cycle. If there is a small drift this will accumulate, then after a few iterations, the error would be enough to be seriously inaccurate. It is important to analyse the timing behaviour to avoid using timers that over time will drift relative to one another. A small inaccuracy in a single cycle might be acceptable, but if it accumulates over multiple cycles, it will be unacceptable.

LESSONS LEARNED & FUTURE APPLICATIONS

The reader will probably be aware that ESA is currently developing the EGS-CC (European Ground System Common Core). At the time of writing development is due to begin, based on the outputs of a design stage. The currently documented design does not give much detail of the command verification implementation, but rather a set of requirements similar to the SCOS2000 functionality. A possible concern might be that this may produce a result similar to SCOS2000, which is acceptable of course, but evidence is currently showing is not flexible enough for the latest missions. In the same way our own produce CCS5 initially emulated SCOS2000, and so the implementation was somewhat similar to that of SCOS2000. We realised that we needed to provide direct access to the internal TC verifier command list, so that we could use scripts to read and modify it. However as we have moved to even more flexible use cases, we have asked ourselves whether the verifier could reasonably have been implemented in scripts from the beginning, thereby having even less, truly minimal behaviour coded in C++. With hindsight we would probably have taken the fully scripted approach.

Another insight is that these new services are a sign of the maturity of the Packet Utilisation Standard. The services foreseen at first definition of PUS, which previously were not implemented because they were "too complex" are now being realised in missions. While the PUS has been quite successful, it has many acknowledged flaws; it is far too loosely defined, and permits so much tailoring that one may ask whether it is really a standard. It would certainly not be possible to directly plug a ground system application from one mission into another. ESA and other agencies have been discussing standards such as CCSDS-MO for some time, but for whatever reason, this does not seem to have achieved a huge momentum or enthusiastic adoption. The interaction between ground systems, on board software and the protocols used between them are fundamental to how satellites can be operated and how their software can develop in future. We think it is rather surprising that such an extensive effort is being put into EGS-CC without a clear definition of what is planned to replace PUS. If EGS-CC primarily supports PUS, and its development consumes most of the available R&D resources, then EGS-CC may unintentionally prolong the use of PUS.

CONCLUSIONS

A key economic issue is that an implementation needs to be highly flexible, and avoid hard-coding as far as possible, while maintaining good runtime performance. CCS5 has achieved this by using the TOPE scripting language, extended to give direct access to the in-memory command cache. With hindsight we would probably have relied even more on scripting than the actual current solution which derives its core concepts from SCOS2000.

The core capabilities are remarkably simple; to be able to revive the verification of a stored or cyclic TC, and the ability to "pretend", or model that we sent a command which was actually sent by some other party. We only require information about its mnemonic and parameters. The most basic use cases that can be scripted in TOPE, are already being delivered to current missions.

There is strong potential to manage much more advanced scenarios, for example to monitor commands autonomously emitted by the satellite itself. However to achieve this would require closer integration with third party tools, such as simulators, emulators or procedure management tools. We should also be cautious not to create an over-engineered solution for marginal benefits. To go further in this direction would require definition of efficient external API's

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.