The Future European Space Automation Domain

Salor Moral, Nieves[1], Dionisi, Simone[2].

[1]) VitrocisetBelgium Spr, Madrid, Spain
n.salor_moral@vitrocisetbelgium.com

[2]VitrocisetBelgium Spr, Darmstdat, Germany
s.dionisi@vitrocisetbelgium.com

INTRODUCTION

Since January 2015, the second phase of the European Ground Systems Common-Core (i.e.EGS-CC) initiative is taking place. This second phase is focused into the development of those components constituting the kernel of the initiative in order to reach the operational status of the developed software so it can be used first in the Neosat mission. Within this set of kernel components, automation will be needed during spacecraft AIT (i.e. Assembly, Integration and Test) phase to perform automated and deterministic testing; and during spacecraft operational phase in order to provide autonomous monitoring and control of spacecraft and ground infrastructure. This paper describes the development done and the resulting system in the automation field under the EGS-CC initiative.
Automation concerns the execution and debugging of activities implemented as Automation Procedures (i.e. AP) or Automation Scripts (i.e. AS) and defined as Monitoring and Control (i.e. M&C) Definitions. As such, the model containing those definitions, the Monitoring and Control Model (i.e. MCM) is in charge of invoking the activities, monitoring their progress and control their executions.
In the context of trying to support space systems monitoring and control in pre- and post-launch phases for all mission types a new European initiative to develop a common infrastructure has been undertaken: The European Ground Systems – Common Core (EGS-CC). The essence of EGS-CC is a common infrastructure forming the basis for the various monitoring and control systems for the different mission phases and for all mission types. This ensures uniformity across the development cycles and supplier chains throughout a mission, and allows for maximising synergies and transfer between projects and project phases, as well as the sharing of maintenance cost.

AUTOMATION COMPONENT

Automation concerns the execution and debugging of activities implemented as Automation Procedures (i.e. AP) or Automation Scripts (i.e. AS) and defined as Monitoring and Control (i.e. M&C) Definitions. As such, the model containing those definitions, the Monitoring and Control Model (i.e. MCM) is in charge of invoking the activities, monitoring their progress and control their executions. Using the M&C Definitions the MCM will determine where the activity will be executed and make the Activity execution request accordingly as illustrated for telecommands and automation procedures in the following diagram.
Although the MCM invokes the activities, the kernel component in charge of executing them for procedures and scripts is

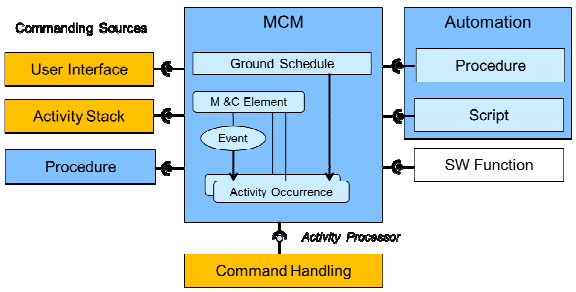

Fig 1. Execution flow diagram

the Automation component, which is the main purpose of this paper. This component is designed as a distributed system, mainly changing between a Controller (which is in charge of receiving the request and notifying the results) and an Engine (the subcomponent executing the activity) which are explained in further detail in the following section.
Due to the need to execute both Scripts and Procedures, the engine has to cope with both execution types in a transparent way to the caller. As such, the system is highly customizable in terms of implementation modifications within the engine because AUT basically calls a third-party engine, that can execute scripts in any language. The only required aspect is that it provides the required interface (at writing time this integration is done for groovy scripts).
On the other hand, the automation procedures executed on ground have a common AP Exchange Format capable of expressing the required features of [1], section 5, plus specific extensions, which was prepared during the first phase of the initiative. This common format, ensures inter-exchange-ability between the various target systems (e.g. from suppliers to customer of a space system or from tests to operations). Moreover, the AP files are always exchanged (between applications, between instances or between environments and missions) on the basis of such format; although APs may be prepared either by the AP preparation tool provided by the EGS-CC, or any other tool capable of generating Automation Procedure Exchange Format like notepad.
At writing time, the AUT component is being integrated with the rest of the core components and demonstrating tthe correct execution of procedures and scripts triggered from the MCM, directly from the UI and from embedded calls in other procedures, thus satisfying the process described in the diagram above. However new capabilities are being still implemented and updates are performed.

Component Building Blocks

As explained above, the AUT component has to satisfy several requirements for providing a deterministic and automated execution of activities. Whereas there are other satellite features, the most important and architecture determinant are:

1. The AUT component has to be able to execute APs defined in the AP Exchange Format and AS required either directly by the user interfaces (UI) or by the MCM.
2. The AUT component has to be able to control running APs (i.e. abort, stop).
3. The AUT component has to be able to debug APs and AS.
4. The AUT component has to monitor the activity occurrences
5. The AUT component has to be able to validate and compile the APs including consistency, type and existence checking.
6. The AUT component shall handle the priority execution of activities
7. The AUT component has to handle the navigation of running procedures (i.e. provide and report dynamically tracing information)
8. The AUT component can make use of the rest of the EGSCC components but cannot fail if they are not present.
9. The AUT component has to report and provide feedback of the execution to the invoker of the running activity.

Based on the previous requirements, a modular service-based architecture was designed splitting the functionality of the validation and compilation of APs and ASs from the real execution, promoting two differentiated deployable units.
The following subsections explain each of the building blocks and its inner components that are shown in the image below (Fig. 2).

Automation Controller
Requests for performing the same functionality (i.e. execution and debugging of activities) can come from different callers (i.e. UI or MCM) based on the general EGSCC system architecture. As a consequence, the component called AUT Controller harmonizes the requests and passes them transparently to the Execution Engine component which is where the execution happens.
However, callers also expect to receive feedback of the sent requests. As the requests are harmonized by the component and the engine is not aware of who has requested what; the controller component receives from the engine the feedback which is forwarded to the correct caller by a mapping stored in the system linking the caller with the unique execution request.
Each type of caller provides specific methods to notify different types of feedbacks such as status updates, error, anomaly or even navigation ones. Through the mapping performed by the Automation Listener Discovery Service, the update is sent via the correct method.

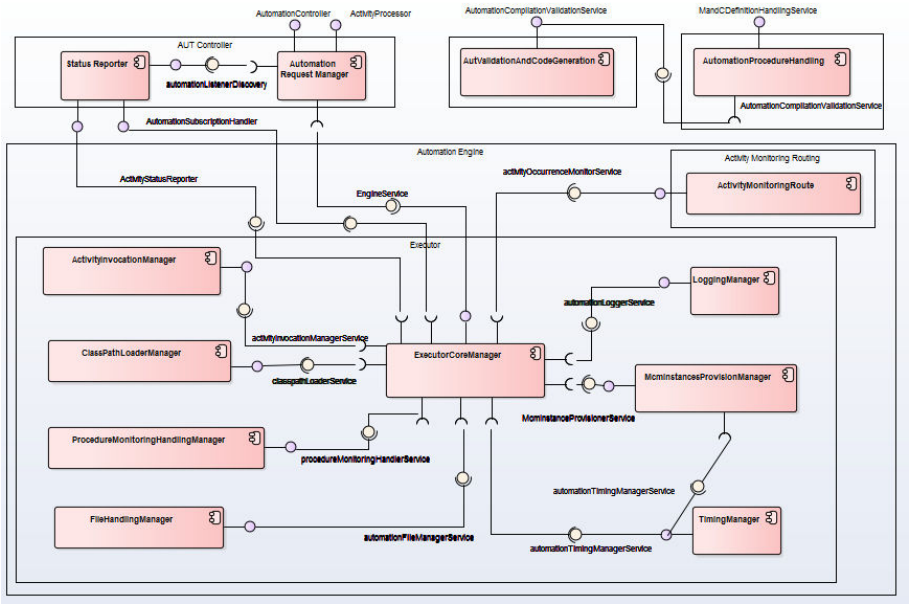

Fig 2. Automation Architecture

Execution Engine

Once the requests have been harmonized, the execution takes place in the core of the system which is the Execution Engine. As the received requests do not distinguish if the activity to be executed is a procedure or a script, this component have to handle both executions in a transparent way.

As such, the first thing to do is to distinguish how the activity is mapped to a concrete implementation (i.e. AP or AS) in order to address the correct behavior. This discovery is performed through a factory pattern by checking against the runtime definition, the type of MCM Activity:

- On one hand, in case the Activity is of type Script, the system calls the provider of the scripting service which compiles and run the specific script. For example, if the script is compliant to the groovy language, it is compiled and run by the Groovy Script[2] which acts as another OSGI service.
- Whereas on the other hand, for procedures, and although they are Java Procedures, they cannot be executed directly by the JRE. The reason for this, lies in the format and monitoring the procedures relies on. Procedures are not isolated classes; they contain references to MCM runtime objects and concurrently can be monitoring/monited by/other processes. This complex process, requires an extra layer handling all the supporting behaviours and error management to avoid fatal consequences in case of execution errors. This extra layer is the constituent of the Procedure Core Engine.

a. Procedure Core Engine

Any execution of a Procedure starts by an invocation. As such, any action for monitoring, controlling or executing any procedure shall be initiated at the invocation request handled by the Procedure Core Engine. This component shall retrieve the specific procedure code which implements the required MCM activity and after found, tell the JVM to execute the Procedure class, following the flow described in part IIIC.

Besides the initialization of the execution of an AP, the component has to answer to the control requests for already executing APs. As the users can decide to explicitly abort or stop a running AP; the Procedure Core Engine will receive these request at the same time it is receiving requests for starting new executions.

As the answer time of the engine is of great importance, each invocation request to the engine triggers the creation of a new thread in charge of executing the specific single procedure (uniquely identified). As such, when a control request is received by the engine, the main thread will look for the thread in charge of that execution and forward the controlling request without affecting the other running procedure occurrences.

The technology used in development enforces a limit on the number of concurrent parallel threds, and thus, number of parallel executions. However, this limit does not constitute a problem as the running requirement express only a need of 50 parallel executions whist the technology limit is of 2025.

b. Script Core Engine

Similar to the Procedure Core Engine, the Script one has to handle the execution and the control of groovy scripts. Nevertheless, this layer is not the responsible of the execution of the scripts itself, but a mere intermediate. Wishing to be able to integrate new scripting languages and/or already existing scripting engines (such as third-party ones), the script core

engine shall simply call asynchronously to the third party engine passing the script content as a groovy file or as a string to let the remote engine to execute them.
In the default configuration, the third party engine responsible of executing the scripts is [3], which comes already intergrated in the Eclipse Environment used as IDE of the EGSCC. However, some processing is done by the component to monitor the different requests and to receive the notifications of the third party engine indicating the problems, the end of the execution, or any feedback neeed. For such cases, the core engine shall act as a listener subscribed at the moment of the invocation, thus closing the communication and forwarding any feedback back to the caller.

Validator and Code Generator
Validation of the MCM information is needed prior to execution for a double purpose. First, to avoid execution of non-valid activities and second to discover the impact modifications can have over other elements. For example, if an activity uses one parameter and the type of the parameter is changed, the activity referring it will not longer be valid if unchanged.
As other components of the EGSCC dealt with activities in abstraction, the only way in which they can be validated is if the specific implementation mapping is validated. Therefore, the system in charge of dealing with scripts and pocedures is the one who has to validate them and provide the results to the MCM.
The Validator and Code Generator component provides answers two main features:

- It is the responsible of providing the dependencies tree and the validation results of the APs and ASs.
- It generates dynamically the code for the MCD elements which can be references inside APs and ASs. This second process is described in chapter III.0.

Development Process

The EGSCC followed several phases where the requirements and the initial overall system was designed so the needs, objectives and interfacing between the different core components would be clear. Afterwards, an agile-like approach for the development of the automation component has been ongoing.
As it is normal, during the development, modifications and updates have been taking place. While most of the updates have been due to technical limitations, integration needs or aspects not previously considered; updates on the design have been performed and the automation component is no exception to this.
The development of the AUT component started with the design of the AP Exchange Format. The definition of the procedure language took into consideration the stakeholder user requirements and [1] and passed over several iterations with the final users until a relatively stable version of the language was agreed between the parties.
During the iterative process, the need of type casting the referred elements arose by part of the stakeholders so type checking could be done at compile-time checking instead of at runtime. This new requirement provoked the need for a new process which would generate a class per data definition contained in the MCD so later could be used in the procedures and a new service, as well, to obtain the later generated set of MCD class sets.
Once the AP Format was discussed, the requirements the AUT component had to satisfy were organized based on priority and dependability between them and the architecture presented before was designed marking the dependability between the subcomponents.
As AUT is a complex system which needs information from other components; instead of starting from the inner-most components the development started in the subcomponents interfacing with external ones. This decision was taken so inconsistencies, gaps, or missing aspects were discovered at the beginning instead of in a later stage where modications could have a bigger impact. This decision proved to be correct as several external interfaces had to be updated and the MCD model has gone through profound refactoring.
With the execution core isolated from outside dependencies and modifications, the development of the core took place next. In this development phase, the engine layer was implemented as a factory/dispatcher pattern in which the script engine relies in a third-party product (i.e. [3]) and the procedure one in a custom one based on the AP execution flow and supporting libraries.
After demonstrating the correct execution of simple procedures and scripts, the complexity of allowed features inside procedures, such as embedded calls or supporting libraries functions was developed and integrated in the system.
After writing time, AUT will continue the development by integrating new features such as debugging following the same process. First developing of the functionality, then unit testing it, third validation testing and fourth integration one. And whenever, some error occurs starting the cycle again.

Testing Performed

During the initial phases of the EGS-CC Initiative, criticality levels were assigned to the different core components of the project depending on the consequences a failure of the component can have. As the goal of the Automation Component is to start and control the execution of the activities not only in AIT phase but also in Operations, a B criticality level was assigned to AUT. This means that 80% of the code lines and branches have to be unit-tested. Besides this, the code has to comply to some coding standards and check-styles.
The testing coverage has been performed using Junit as testing framework supported by Mockito tool and monitored by the use of the automatic integration testing tool named Jenkins and monitoring one named Sonar.
After the code has been successfully unit tested, the components have to be validated in an integrated environment. This validation process takes place during development time, so the real components with which the integration takes place are still not available. The validation tests are responsible of proofing the developed system satisfies the component requirements. For doing this, use case scenarios have been described and coded linking them with the requirements that each scenario demonstrates.

Up to this stage, the testing has been performed by the development team of the AUT component and real integration with the rest of the components has started to be performed. This phase of the testing is performed at the end of each sprint by the consortium IVT team which has the final word about the validity of the component.

Expected output

At writing time, the integration of the Automation Component has been performed successfully with the other EGS-CC components after several iterations. However, the level of functionality provided at this first integration does not satisfy all the requirements the component has to provide.
During the last months, several features have been developed and tested including three important ones:

- Debugging of APs and AS using the Java and Groovy Debugger and displaying the debugging process.
- Logging of the information for the creation of dynamic as-run reports of each activity execution
- Verification support library for parameter values.

While some others still have to be implemented, such as:

- Inclusion of the monitoring information of the procedure execution
- User Interaction from/to procedures.

Once all the features are developed and integrated within the EGS-CC, the automation component is expected to control the execution of the activities in both Operations and AIT phases of the future space missions starting by NeoSat [4] in 2018.

APS HANDLING

In this chapter a detailed description of the capabilities and format of the EGS-CC procedures is presented in order to show readers the high extensibility and low maintenance the automation field is targeting to.

What is an AP

Procedures are a subtype of MCM activities, but also a procedure is an activity embedding other activities to be executed within the execution frame of the procedure.
Technically speaking, an automated procedure is a piece of code written in high level code following the JAVA programming language. As such, through a procedure the programmer has the capability to use the full JAVA capability to automate functionalities and access to the EGS-CC Kernel main components through call to OSGI services.

Integration of space domain concepts in the Aps

Dynamic Code Generation
Usually, Automated Procedures in space involve verifying telemetry data and sending telecommands to the different SUT involved in a mission. As such, any of the referenced TM/TC appears in the APs as typed elements. In EGSCC [5] the possible types of the referenced elements are understood either as MonitoringControlElements (MCElement from now on)

or as the MonitoringControlElementAspects of an MCElement (MCAspect from now on). Within the system, the term MCAspect abstracts the concepts Activities (e.g. TC), Parameters(i.e. TM) and Events.
Due to security reasons, it is equally important that the system prevents users to incorrectly operate with MCElements or MCAspects (e.g. pass an argument value inconsistent with the required data type) than the way to write procedures is user friendly and understatable for non-programmers. As a consequence, the system has to provide compile-time checking for several situations, although the most relevant are:

- Provide access only to the types of MCAspects defined for an MCElement
- Consistency checking of the Raw, Source and Engineering Data Types, Values and Units for Parameter.
- Provision validation of the mandatory, optional arguments of an Activity MCAspect.
- Consistency checking of the values and types of the arguments provided for the invocation of an Activity.

For supporting this compile-time checking, at the start up of the system and each time there is a modification in the set of definitions of the MCD; a process in charge of dynamically generating the class type for each type of referenced element is called.
This process, named the MCD Code Generator, takes the set of definitions and after checking the type of definition it generates a class per definition (but for parameters) which extends an existing template class. The template classes make use of Java generics which are customized for each of the definitions, thus specifying the required data types, aspects and units needed for each definition.

Dynamic Use Of The Referenced Concepts

Although the dynamic generation of defitinion classes allow the users of procedure to limit potentially fatal error (e.g. mistaking unit systems) at edition time, the classes are empty shells that have to be filled with runtime information of the instances compliant to those definitions that are populating the domain. It is important to remark that the same concept definition can be typing several runtime MCElements and MCAspects. And as such, each of the instances will have different live value data all compliant with the data types of the definitions.
The Automation System is the responsible of initializing all the intances of the referenced concepts of a procedure when it is invoked. As the runtime instances are unique, the initialization only has to happen once, but when it is done, the runtime concept has to start receiving live information for its specific requests. For examples, parameters shall start loading the raw and source values received, the elements will discover which specific events are happening, or which of the activities defined are being executed, etc. Nevertheless this process is completely transparent to the Automation User as they will just use the referenced concepts and perform operations with them.

Execution process of an AP

For executing procedures, a set of pre and post processes have to be performed to allow the dynamic control of the execution from remote, the error management and the monitoring of the MCM referenced elements. As a consequence, when a procedure is executed the following processes take place:

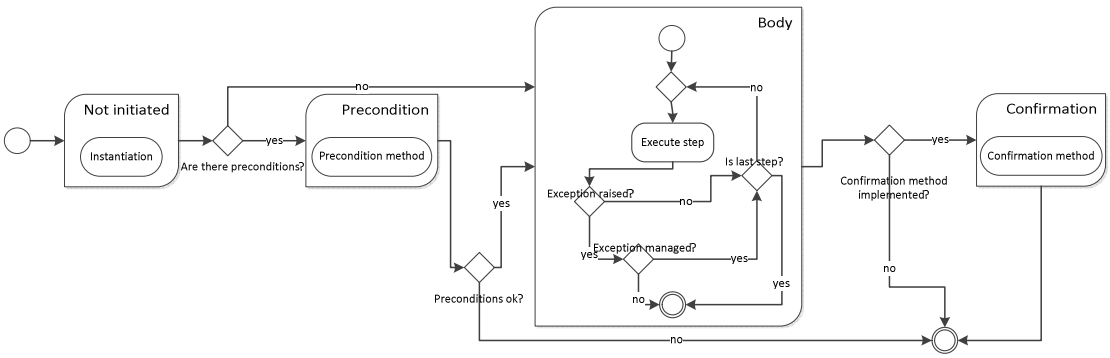

Fig 3. AP Execution Flow

- A unique runtime ID of the activity is created linking it to the runtime MCM object and stored in an internal service for control.

- The procedure implementing the invoked activity is retrieved and compiled (if not already).
- An instance of the procedure object is created using a classloader containing the latest generated MCD Set.
- A runtime annotation processor is run to discover the referred elements inside the procedure:
    - If there is some annotation, an instance of the field annotated is generated, its value is initiated with the runtime value of the MCM element matching the annotation value and the field value is injected in the procedure instance. In the initialization, the registration for receiving updates of the vaues (e.g. for parameters) is performed as well.
- After the element initialization, the procedure is executed and controlled by the execution layer according to a tailored execution flow of the one defined in [1]. This new flow has been generated due to the experience gained from the years of applying the standard, in which some circunstancies were not used frequently by theoperators. The new flow is summarized in the diagram above.
- During and at the end of the execution, feedback is sent to the caller user through injected callback and information is stored in logs to later be filtered depending on user criteria to generate as-run reports.

Java APs

AP Exchange Format

Procedures for EGS-CC are based in Java Procedures, as such the AP exchange format is an extension of Javain which certain functionalities are provided by default to the users. These functionalities are classified based on their feature goals. For one side there are those methods which expose and handle the MCM Element referenced in a procedure (i.e. parameters, activities, events, elements and arguments); and on the other, those general behavior statements which provide control mechanisms to the user (e.g. waiting for durations, for Boolean conditions or for timestamps; abort method or request input information).
Any AP extends the abstract class named Procedure which provides a default implementation of the structure explained in **¡Error! No se encuentra el origen de la referencia.** and [6]. This default implementation is shared with the Automation Preparation Environment, so users can create/edit their own procedures with all the edition support any Java Editor provides.

Supporting Libraries

As a full programming language is not user-friendly for different stakeholders, such as scientifics. a set of classes and pre-defined supporting libraries which provide access to the main EGS-CC Kernel components (e.g. M&C Access API, FIM API, SDA API etc.) in order to allow control over the EGS-CC system have been defined.
Organisations using EGS-CC have the possibility to change and replace existing support libraries with more specific related functionality e.g. to analyse and manipulate packets. This allows the flexibility to freely improve the library by the different organisations but would of course limit exchange of procedures as both sides would have to have the same support library installed. In order to have a consistent behaviour in the execution, the Support Libraries have to be delivered together with the Automated procedures and included in the Execution Environment.
By default, the Execution Environment provides supporting libraries for handling times, dates, files management, logging capabilities, calibrations, monitoring checks, packets, report generation, encoding and decoding and bulk approval of dangerous activities.

Error Management

Exception is an error event that can happen during the execution of a program and disrupts its normal flow. Java provides a robust and object oriented way to handle exception scenarios, known as Java Exception Handling.
Java provides specific keywords for exception handling purposes (i.e. Throws, throw, try, catch, finally…). Whenever an error occurs while executing a statement, an exception object is created and then the normal flow of the program halts whilst the JRE tries to find something that can handle the raised exception. Whereas exceptions are not handled by the procedure programmer, they will be caught by the execution environment and will cause the procedure to be aborted.

However, this approach is currently undergoing some modifications for some crucial statements such as activity invocations, parameter forcing values and value verifications. However in normal cases, the exception object contains a lot of debugging information such as method hierarchy, line number where the exception occurred, type of exception, step that generates the exception, etc. which is useful for replaying or for the operators to trace issues that have arisen during the procedure execution at AIT phase.

Nicer Procedures are possible

The use of any software programming language for the procedure definition, in AUT case Java, assumes final users have experience programming in such language. This does not happen in most of the cases where operators or final users do not have to know Java and, furthermore, some users such as scientific ones, may not have any programming skill at all. As described in section 3D, Java is not even applied in its basic stage because new capabilities (i.e. annotations and lambda expressions) that have appear in Java 7 and 8 are used. These new aspects of the language are not trivial even for an experienced programmer; thus, the target stakeholder set of the EGS-CC may be deeply reduced.
In order to improve the user experience and to allow reusing existing procedures written in other languages or formats, a layer on top of the one provided by the core of EGS-CC is required. This layer is understood as a reference implementation and at writing time is being developed as well under the name of OPALE [9].
In fact the term "inter-operable" used in the name of the project refers to be capability of executing automatable procedures prepared with different languages within an environment that will run them using one common execution model as if all procedures had been prepared using the same language. These multi-input format procedures will be automatically transformed in a transparent way into the EGS-CC AP format required for the execution or debugging of activities by the AUT component.
The execution and/or debugging of a procedure is traced back to the format in which the user has defined the procedure and not to the AP one. Therefore, there is a connection between the preparation environment with the execution one of the EGS-CC which includes all the needed information for showing the executing/debugging information to the client within the DSL editors. As a consequence, the final user does not need to be aware of the EGS-CC running underneath of its normal automation environment.

STATUS AND FUTURE STEPS TO BE TAKEN

Although the Automation field is complex in its own, the development of the Automation Component within the EGS-CC is proving to be challenging due to the huge amount of stakeholder needs, the integration with several other components, the deep and wide knowledge required to understand the interfacing required and the distribution and amount of people working in the consortium.
In the same way, this development is providing the involved team with a wider and deeper understanding on all phases a mission goes through and the real needs of the stakeholders; which sometimes get lost in the communication chain.
At writing time the development of the Automation Component is not complete, but it will be during 2017. In parallel, development on top of the EGS-CC with Reference Implementation Systems is foreseen to be carried-out in terms of developing languages and transformations on top of the APs, as done in the OPALE project explained in section 3E or developing middle layers between legacy systems and the EGS-CC to allow the execution of existing automation systems together with the EGS-CC.

REFERENCES

[1]ECSS-E-70-32C - Ground systems and operations-procedure definition language; issue 2.0,July 2008
[2]Groovy,*"Groovy Script Language"*, Open Source, supported by Apache Foundation, version 2.2.1, 2016
[3]Groovysh,"Groovy Shell", http://groovy-lang.org/groovysh.html
[4] NEOSAT, "*Next Generation Platform*", ESA,
http://www.esa.int/Our_Activities/Telecommunications_Integrated_Applications/Next_Generation_Platform_Neosat
[5] Walsh,A; Pecchioli, M; Charmeau, MC; Geyer,M.; Parmentier, P; Rueting, J; Carranza, JM; Bosch, R; Bothmer, W.; Schmerber,PY.;Chiroli,P. "*The European Ground Systems Common Core (EGS-CC) Initiative"; 11-15 June 2012. SpaceOps2012.*
[6] EGS-CC System Engineering Team, "*Automation Procedure Programmer's Guide*", ESA (unpublished)