EKSE: A Command Line Interface for EGS-CC based Systems

Tom M. Schiller[1]
European Space Agency, ESA/ESOC, Robert-Bosch-Str. 5, 64293 Darmstadt, Germany

Until recently, the interaction with systems based on EGS-CC (the European Ground Systems Common Core) was limited to the graphical user interface developed as part of its reference implementation. However, the need has been identified for an interactive console to interface the system. Several approaches for such an interactive console have been considered and will be introduced in this paper. The approach finally chosen is referred to as EKSE (EGS-CC Karaf Shell Extension), which enables users to define commands based on scripts which can even be created at runtime, enabling fast prototyping for interactions with the backend. Particular challenges will be highlighted and approaches for meeting them will be proposed.

I. Introduction

The European Ground Systems Common Core (EGS-CC) initiative is in a phase of active development. The goal is the creation of a set of components which make up a fully functional system core and a strong reference implementation. Based on this common set of components, various monitoring and control systems for missions in their pre- and post-launch phases and even for ground stations can be composed all throughout Europe. [1] To compose such systems, new components can be added either on top of the existing system, or can replace components of the reference implementation to add new functionality. Several stakeholders have already started this process of creating their own M&C systems for both spacecraft and ground stations, which is referred to as the integration process. It is based on regularly published Integration Releases of EGS-CC, which give a preview of the capabilities of the system that is being developed in the context of collections of components that are integrated into a Test System Instance (TSI). At ESOC in particular the ESA Ground Operation System Common Core (EGOS-CC) is developed by integrating additional ESA-specific components with EGS-CC. [2] This forms the basis for full systems that will eventually be ready to be used operationally, one particular example being the mission control system for the Jupiter Icy Moons Explorer (JUICE), which is currently planned for launch in 2022. [3]

A lot of integrator activity during this phase is focused on testing the delivered EGS-CC based Test System Instance and these tests can be quite involved and take a long time. This leads to the need to automate at least some parts of these test runs. However, the interaction with an EGS-CC based system as delivered originally is only supported through the graphical User Interface component (UIF). It is part of the Reference Implementation and shows how an end-user, such as a spacecraft controller, could interact with the system. The graphical nature of this tool means that automating any interactions with it is quite difficult and inconvenient.

Testing is also made more difficult by the fact that so far no single part of the system is entirely reliable: If problems occur, there is no immediate way to know whether they occur inside one of the core backend components, one of the components of the Reference Implementation, or inside the User Interface itself.

The ever-evolving state of the core system, which changes a lot from version to version due to it still being under active development, complicates integration activities as well. As this makes it inconvenient to produce detailed documentation on technical details which might be outdated by the next release, a lot of the documentation foreseen for the final release is not yet present, which necessitates a lot of exploratory development on the side of anyone trying to build new functionality on top of this system. [4] This can be most easily accomplished by using a running instance of an EGS-CC based system and iteratively trying out different versions of the same source code, observing whether they perform the required actions or not, such as calling data-supplying backend services with differently specified filters to practically understand how the filter fields are interpreted. However, the default way of deploying source code in EGS-CC is by rebuilding the changed components and restarting the entire system, as dynamic

[1] Young Graduate Trainee, tom.schiller@esa.int

reloading of components is not yet supported in all cases. This is very inefficient, as restarting the entire system can take a long time.

Attempting to solve these problems, it was decided to create a new interface for accessing EGS-CC in order to simplify the integration and validation tasks at ESOC.

II. Requirements

Before being able to develop the new interface, we needed to clearly identify what kind of an interface was actually required. In particular, we needed to decide on who the intended users at various stages in the life cycle of the EGS-CC project itself would be, and then understand their individual use cases.

A. Graphical or Command Line Interface

We first of all noted that the main reason why the existing graphical UIF component was not sufficient for the ongoing integration activities was its inherently graphical nature, as this made it difficult to automate interactions with it and prevented users from quickly redefining the contained source code due to its immense complexity.

Ideally, we wanted to create an interface that allows simple interactions with a running EGS-CC based system, in addition to enabling the user to change the executed source code while the system continues to run, such that for one particular system start many different source codes can be tested and their actions observed. This necessitates the split between the tool itself, which provides just the environment necessary for executing generic source code within EGS-CC, and an amount of predefined source code snippets which can perform certain actions, but which can be redefined and even completely replaced on the fly. We refer to these source code snippets as ‘scripts’, as their dynamic execution is similar to the truly dynamic interpretation of scripting languages. It should be noted though that this is just how the behavior presents itself to the user; internally, the scripts are loaded upon the user’s request, compiled, and then executed together until the user requests again to repeat this process.

B. Target Users

So far EGS-CC is mainly used for preliminary testing, familiarization of future system administrators and technical officers, as well as early developments on top of it. Therefore, the focus of the command line interface lies on providing for the needs of integrators who are performing tests of the integrated Test System Instance and the needs of developers of future components for EGS-CC based systems.

However, this focus on the current needs does not prohibit the newly developed command line interface from also being useful once the core EGS-CC development has been finished. In fact, the potential for aiding future administrators of operational systems has also been foreseen, who could benefit from being able to interact with any part of the EGS-CC backend directly and from being able to automate whichever steps they deem necessary. Therefore, the future use of the command line interface has been kept in mind from the earliest design stages onwards.

Despite this dual focus on both the current use and future use, not all users of EGS-CC based systems are targeted by the newly developed command line interface. In particular, the real end-users of future EGS-CC based M&C systems are spacecraft and ground station operators. These will interact with a whole system built on top of it rather than just with the currently existing Test System Instance, and most likely will be presented with augmented versions of the existing UIF or even with newly developed graphical interfaces specific to the exact tasks they have to perform. These interfaces will also give access to the so-called ‘Automation Procedures’ which are foreseen to enable interactions with the higher-level concepts of the controlled systems, rather than accessing low-level EGS-CC services directly. Such interactions would not be aided by having to go through a dedicated command line tool, which means that future operators using EGS-CC are not the intended target users for the command line interface presented in this paper.

This also works well with the security concept of EGS-CC, in which an operator should have control over the system that is being operated on, but not over the monitoring and control system itself - i.e. the end-user should use the monitoring and control system as a tool, but not modify it directly which could lead to various unexpected problems. A system administrator on the other hand, who has direct access to the backend, is able to perform disruptive actions on the monitoring and control system anyway, as this level of ability is needed for performing the task of administering the system. Therefore, giving such a powerful tool to a system administrator is not opening up additional security vulnerabilities, but merely empowering them to perform their work more directly and efficiently.

C. Command Line Use Cases

The explicit use cases that we identified for the command line interface are:

1) Enabling any user of the command line interface to start using without great difficulty. As EGS-CC itself is quite complicated, it should be as simple as possible to get started with the command line interface.
2) Enabling an integrator performing a test to interact with an EGS-CC based system in a repeatable manner, the outcome of which can be easily shared and stored as it can be copied from the console output.
3) Enabling an integrator performing a test to automate one or several of the test steps.
4) Enabling an integrator performing a test to determine which part of the system is showing a certain unwanted behavior, e.g. a particular backend component or the UIF.
5) Enabling an integrator performing a test to directly interact with local interfaces which are not usually available to external tools, to be able to more deeply analyze the behavior of the system under test.
6) Enabling an integrator or system administrator the execution of basic tasks, such as creating and starting sessions, reading out parameter values and invoking activities.
7) Enabling exploratory development by allowing a user to redefine the commands available within the command line directly at run time, while being connected to a running EGS-CC based system.
8) Improving the speed of new development based on EGS-CC by enabling the recombination of existing scripts as building blocks for more complicated instructions, which ideally can be reused in other contexts.

III. Implementation

Even after it was clear which behavior was required of the future command line interface, deciding upon a particular technology to be used to implement this behavior was not straightforward and we considered several options for creating such a command line interface.

A. Using an Existing Command Line Tool

The first idea was to reuse an existing EGS-CC command line functionality, referred to as the Script Execution Command Line. Even though it was not required for such a tool to be delivered at that point in time, this command line tool had been created already, and enabled its users to call it with particular arguments which were executed as scripts running directly within EGS-CC.

However, this did not offer an interactive environment which would once be started and in which then commands would be executed, but instead for each command the tool needed to be restarted and the connection to the backend established. This made it very simple to automate interactions with the system from a truly external context, e.g. via shell scripts which could even be started remotely. For executing a few commands this was a useful approach, but when trying to automate a lot of commands and call this tool again and again, the performance penalty would have been prohibitive, such that we decided to look for other alternatives.

B. Creating an External Program

We then considered creating a new external program that can connect to EGS-CC, and after starting up allow the execution of several commands independently.

However, at the time of the first Integration Release, delivered in 2016, connecting to EGS-CC from a completely external, generic system via e.g. a web-based REST API, or a CORBA interface, or any other established standard was not supported. Therefore we needed to instead create a program that operated as part of EGS-CC, using the EGS-CC-native internal APIs and connection methods, which are based on OSGi.

C. A Standalone Solution

We then tried to create such an OSGi-based program as a new standalone application, offering an interactive command line interface to the user. We refer to this approach as EGS-CC Standalone Command Line Interface, in short ESCLI.

Due to the choice of OSGi as underlying technology we were restricted to using a programming language running on the JVM, and as EGS-CC itself is written in Java we chose Java for the command line interface as well. After some investigation we discovered that while Java makes it easy to write a classical program running inside of a command line which is started once with a particular set or arguments, it is actually rather difficult to create a program running inside an existing command line and starting an interactive user session in which new commands can be typed in by the user and the program responds with its answer, going back and forth multiple times. Projects

such as jline[2] and Spring Shell[3] exist which enable this kind of capability, however they also show that this is far from trivial.

Another approach would have been to create an inherently graphical interface, which just contains a text field in which the user can input data and which displays output to the user. This approach offers ultimate flexibility, as the information shown to the user can be displayed using fixed-width characters inside the text field, but when necessary also any other kind of element could be displayed, such as opening a hint box as soon as the user hovers over a previously typed command with the cursor.

However, having to create the entire terminal application rather than just a program running inside of an existing interactive console seemed unnecessarily complex, such that we continued searching for other alternatives.

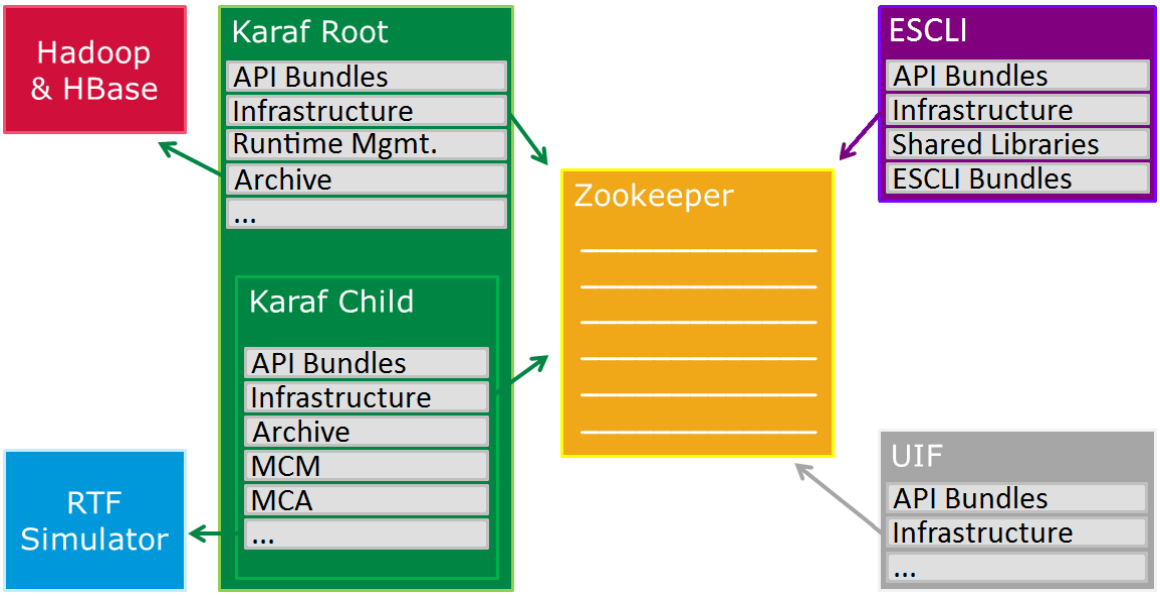

Fig. 1 ESCLI (top right) in the context of a typical EGS-CC based system.

D. Using an Existing Console Application as Host

After further interactions with the system, we noticed that whenever the EGS-CC Test System Instance is starting, several interactive command line sessions were shown to the user. This included a console for interacting with the central Apache Zookeeper server, which is used for the discovery of remote services, and a console for interacting with Apache ServiceMix, which is the container in which the EGS-CC components are running.

We furthermore realized that instead of introducing yet another console application, it would be quite useful to be able to reuse one of these consoles that were started anyway, as this would reduce the amount of windows and therefore the overall perceived complexity by the user. In addition, reusing such a console also meant that we could circumvent the problem of having to create an entirely new terminal application from the bottom up.

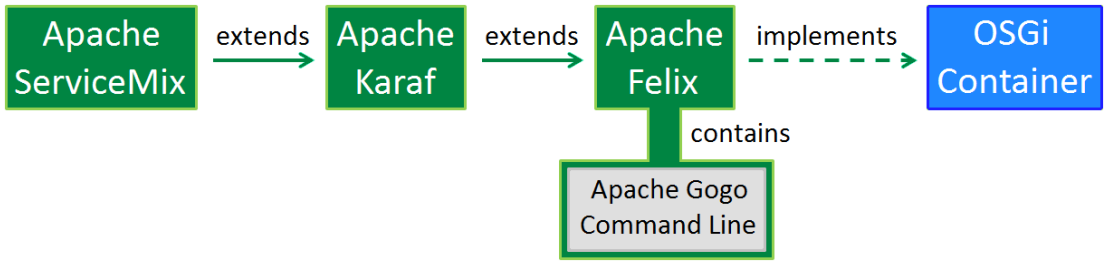

Fig. 2 Schema of Apache technologies often simply referred to as 'Karaf' in the EGS-CC project.

[2] https://github.com/jline/jline3
[3] https://projects.spring.io/spring-shell/

Of the console applications running as part of the EGS-CC Test System Instance, we then chose the ServiceMix console rather than the Zookeeper one, as it provides access to the OSGi container within which the EGS-CC components themselves are already executed. Figure 2 shows a more detailed explanation of the underlying technologies. Usually, we refer to this container as Karaf container rather than ServiceMix container and due to this, we are referring to the resulting command line tool as EGS-CC Karaf Shell Extension, in short EKSE.

This provided us with two candidate applications: A standalone command line interface, or an OSGi bundle enhancing Karaf with additional EGS-CC related commands.

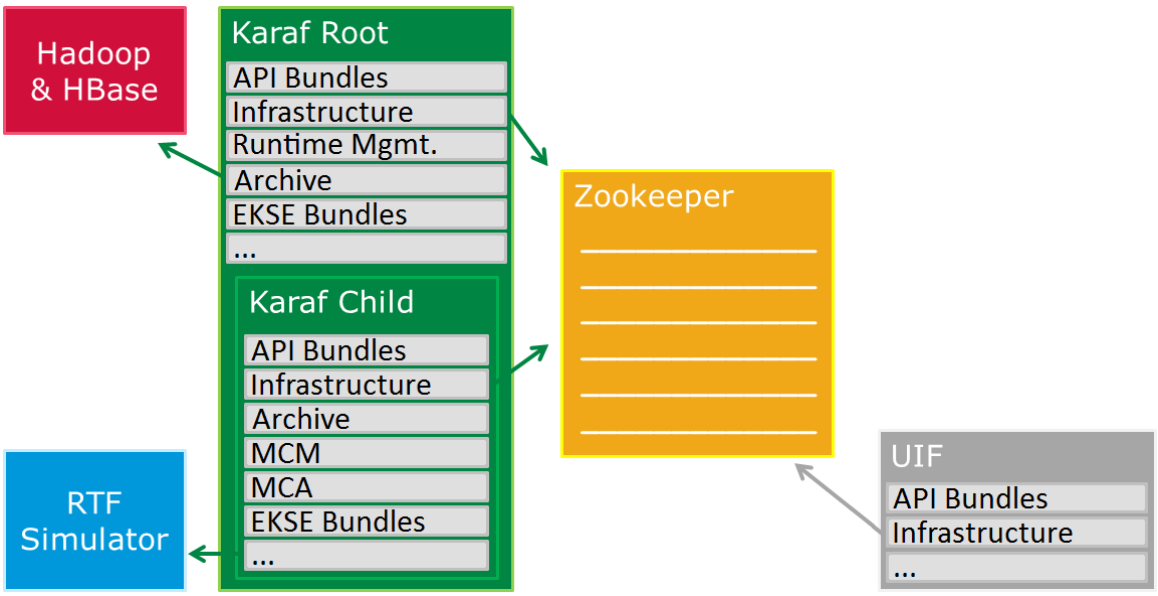

Fig. 3 EKSE (bundles inside Karaf) in the context of a typical EGS-CC based system.

E. Advantages of ESCLI

To decide which approach to use, we consider the advantages and disadvantages of the two possible approaches we have identified.

Starting with the advantages of ESCLI, we can first of all see that the user might have to prefix commands in EKSE in a specific way, as the Karaf environment already contains other commands. E.g. if we were to create a command to list certain items, we could just call it as `:list` inside ESCLI, while there would be two such commands available in EKSE, `admin:list` and `egscc:list` which means that the user always has to use the `egscc` prefix. However, this is not a very important advantage for ESCLI, as the user is allowed to leave out the namespace in the Karaf console as long as the command is still unique, and we can simply define commands with names different from the ones already being defined inside Karaf.

Additionally, in EKSE the parsing of the parameters is performed by Karaf itself. This means that certain letters, which provide further functionalities, cannot be used in a straightforward way. An example for this is the exclamation mark, which has to be escaped in Karaf:

```
egscc:logmsg "Hello world\!"
```

In ESCLI, there would be no need for such special characters, and they could be entered by the user without escaping them:

```
:logmsg "Hello world!"
```

With the standalone approach we are also just in general guaranteed more flexibility; as we are creating an entirely new terminal, we can output data in whatever way we want and are not technically bound to the terminal itself. This is especially useful for providing hints to the user when commands are entered incompletely, by offering to complete the name of the command itself or even by offering additional information about the various arguments that can be used with it. An example for this can be seen in Figure 4.

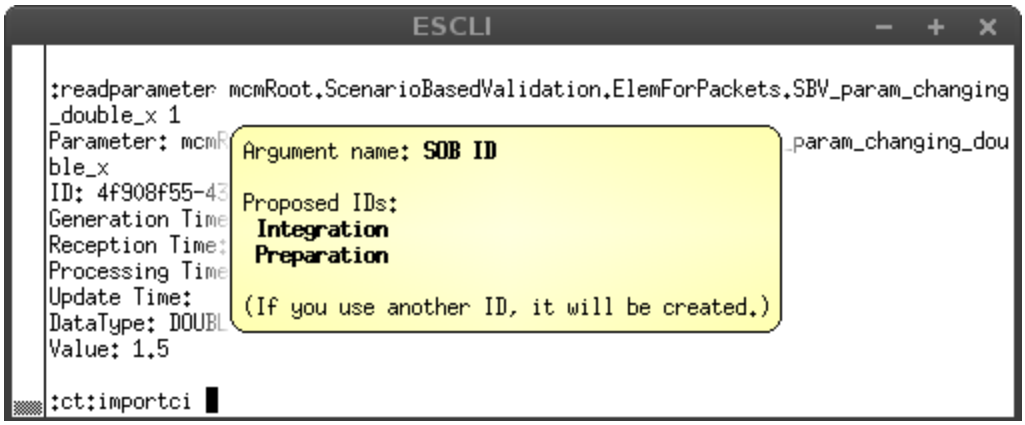

Fig. 4 Showing the potential for ESCLI tab completion info box after entering `:ct:importci` at the bottom and pressing the tab key.

In EKSE, it is also possible to provide proposals for arguments by means of so-called completers, as seen in Figure 5. However, they are restricted to the output methods offered by the terminal application itself.

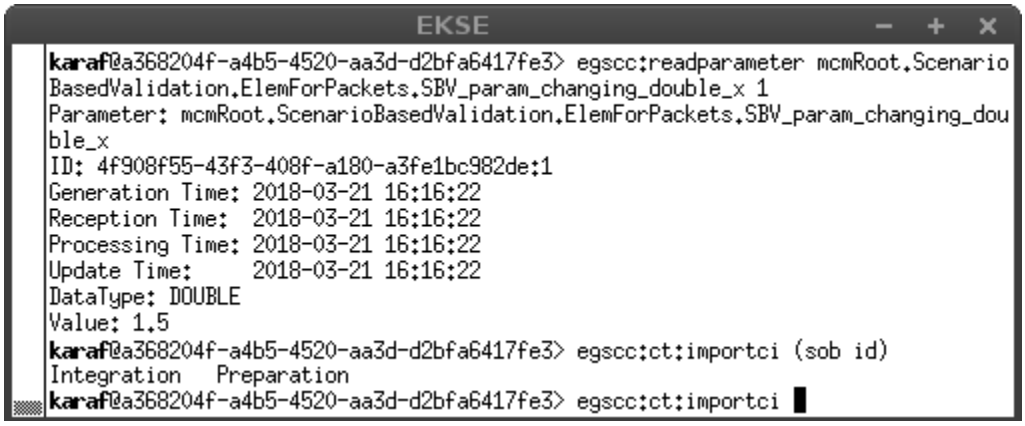

Fig. 5 EKSE showing similar tab completion info, but bound to purely textual interaction with the user.

An even bigger problem with the missing flexibility inside of the Karaf console is that while EGS-CC is still under development, the system is sometimes not behaving particularly well, especially when performing intensive stress tests. This leads to instances of components writing large amount of output to the standard output stream which are shown directly in the Karaf console, or even to exceptions arriving there. These can get in the way of interacting with EKSE. Again, with ESCLI we have no such problem, as we are not bound to any existing output being forwarded to the user.

Finally, when developing EKSE we are tied to Karaf, and if the technology baseline of EGS-CC ever significantly changes and a different OSGi container is used to run the backend, then this would mean that the command line would have to be significantly reworked. Although this is a rather unlikely scenario, it is quite likely that the particular version of Karaf which is in use during the development activities will not be used by EGS-CC forever, but instead newer versions will be adopted over time. Again, if these newer versions were to significantly change their behavior, EKSE would need to be reworked, while a standalone solution such as ESCLI would not have this problem. To investigate whether this is likely to become a problem, we compared the version of Karaf currently used within EGS-CC, which is 2.4.1, with the latest available version of Karaf, 4.0.5. This investigation revealed that the extension mechanism that EKSE is using is still largely unchanged, and seems stable even over long durations of time, such that relying on it is not expected to be too problematic.

F. Advantages of EKSE

Considering now the advantages of EKSE, we first of all note that simply reusing the existing Karaf console means that less has to be newly developed, and a solution which is basically ready off the shelf and already well

tested can be used. This means that the development will be quicker and most likely result in a reliable product sooner than that of ESCLI.

Reusing the Karaf console which is started anyway as part of the system also results in there being fewer distinct programs that users such as system administrators for EGS-CC have to familiarize themselves with. In fact, the Karaf console and the newly developed command line tool may even be seen as the same product from the perspective of the user, which is an advantage as the learning curve for interacting with EGS-CC is already rather steep.

Providing fewer different products also means that less configuration efforts have to be undertaken. E.g. when using ESCLI, it would be necessary to configure the IP address and the port of Apache Zookeeper, while in EKSE no such configuration is additionally needed as Karaf is configured with these values already.

Using the ESCLI approach might also mean an unnecessary duplication of functionalities, as users might expect and demand certain functionalities from the EGS-CC command line that are already provided by Karaf natively. In the case of EKSE, it is simpler to convince a user that such a functionality already exists in the system, as it is in the very place where the user would expect it when interacting with the command line.

Furthermore, the formal integration of ESCLI with the EGS-CC based system would require a small API change in the Level 0 Interfaces of EGS-CC itself. Such a change can of course be performed, but would lead to costs as all developers and integrators involved would have to adapt, such that it is ideally avoided.

Finally, the ESCLI implementation as it was foreseen by us required additional setup code inside the user-supplied scripts to enable output and error forwarding to the command line itself. Forcing the users to provide such repetitive boilerplate code however seems rather inelegant and should also be avoided.

Having considered all this, we decided to select EKSE over ESCI – that is, we decided to enhance the Karaf console with additional EGS-CC related commands.

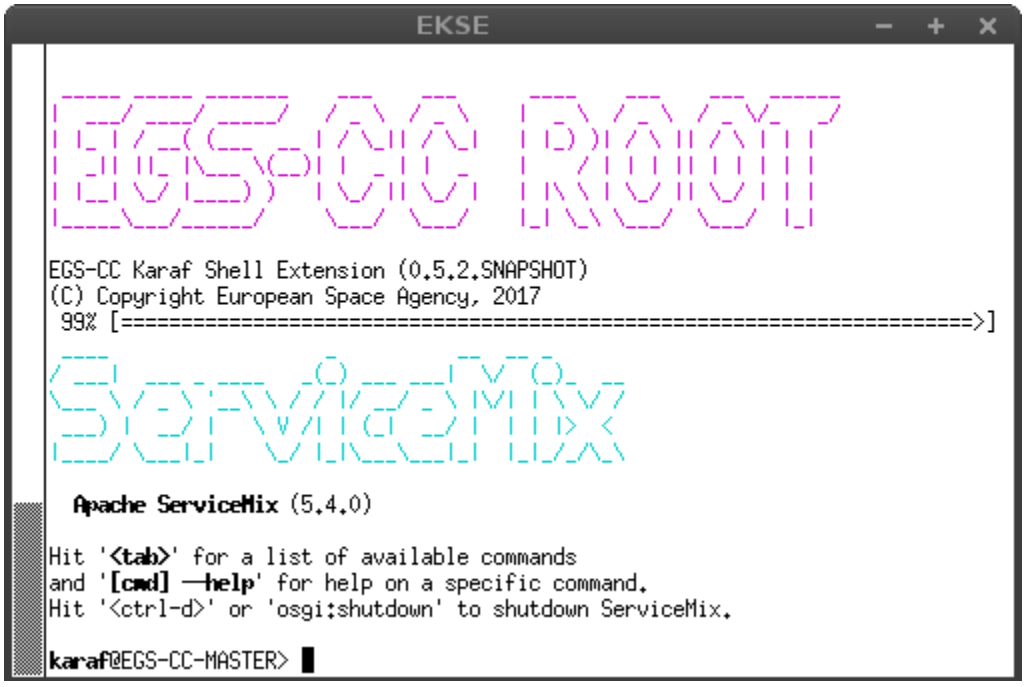

Fig. 6 EKSE inside Karaf / ServiceMix just after startup as part of the EGS-CC IR3 Test System Instance.

G. Choosing a Language

When the choice had been made to let the new command line interface directly interact with the OSGi-based APIs of EGS-CC, it became clear that a programming language running inside the JVM would need to be used. And indeed, within the EGS-CC project, both Java and Groovy are foreseen to be used, with Java being the language in which the backend components themselves are written and Groovy being intended as a scripting language.

The choice of Groovy for EKSE may therefore seem clear-cut, however there were also arguments speaking for using Java, as the created scripts are intended to form reusable building blocks, which could more easily be integrated with future components developed on top of EGS-CC if they were written in Java directly. In fact, the

current idea is to take some of the functionality originally developed within EKSE out of it, and create a Java-based toolbox that can be reused more easily in various contexts.

Due to early considerations about such reusability, the decision was ultimately made to adopt Groovy as simpler to use scripting language than Java, but to try and write it in such a way that a later transition especially of core scripts to Java would be as painless as possible. This can be achieved as nearly all valid Java code is also valid in Groovy, and therefore Groovy can be written such as to be very nearly compatible with Java without any changes at all. This approach allowed us to use the flexibility and simplicity of Groovy when just quickly writing one-off test scripts, but to also employ more rigorous Java-like code for core sections of the scripts that would more likely be reused in different contexts later on.

For an example of an EKSE script written in Groovy, see Appendix B.

H. Enabling Truly Dynamic Interactions

One of the requirements we identified earlier is that it should be possible for the user to redefine commands on the fly while EKSE is running. The underlying technology however makes this rather complicated, as it is assumed that an application is composed of various components which are wired together using a technology called Blueprint.[4] This means that all connections between the services offered by one component and the ones required by another one should be provided by XML-based Blueprint files.[5]

As we are not restarting the entire command line tool every time that the user is updating some scripts, we cannot change the Blueprint file of the tool on the fly to react to such updates, as this file is only loaded once by the framework when the tool is started.

To define the command `egscc:run` in Karaf, which is using the completer `filePathCompleter`, the following would need to be written in the Blueprint file and the entire EKSE bundle rebuilt and redeployed, as shown in Figure 7:

```
<command-bundle xmlns="http://karaf.apache.org/xmlns/shell/v1.0.0">
  <command name="egscc/run">
    <action class="esa.egscc.kernel.cli.commands.Run"/>
    <completers>
      <ref component-id="filePathCompleter"/>
      <null/>
    </completers>
  </command>
</command-bundle>
```

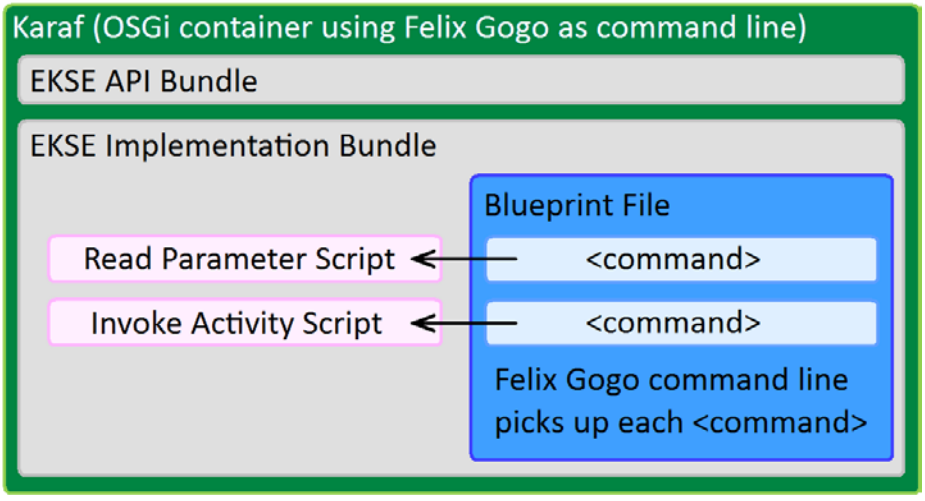

Fig. 7 Possible deployment of EKSE in Karaf with scripts registered using a Blueprint file.

[4] http://aries.apache.org/modules/blueprint.html

[5] Other approaches were considered by the team working on the version of Apache Karaf used in EGS-CC, but not deemed necessary at the time, see e.g.: http://karaf.922171.n3.nabble.com/Simpler-karaf-shell-command-action-definition-td3974398.html

To avoid the need for this overhead, we decided to use a different internal structure, avoiding Blueprint altogether, as shown in Figure 8. This complicates the structure of EKSE, but makes it possible for the user to redefine scripts at runtime without needing to edit any Blueprint files, and without needing to redeploy any bundles.

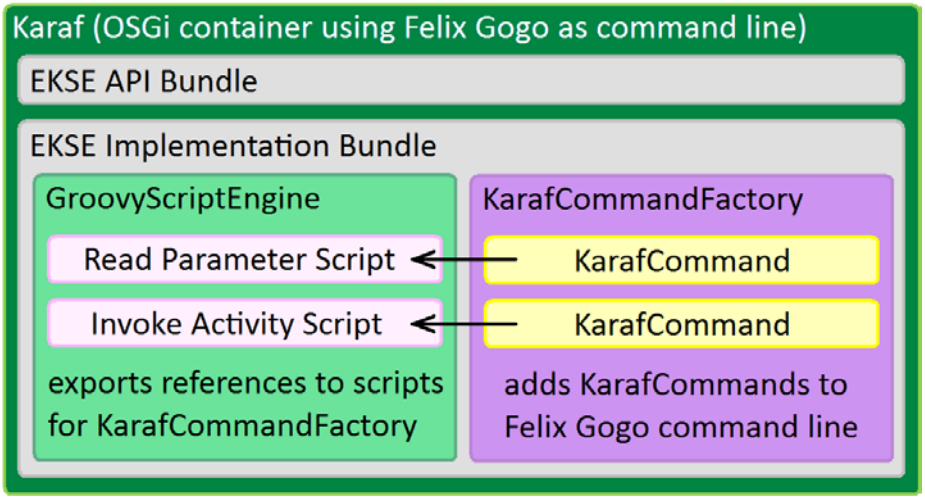

Fig. 8 Actual deployment of EKSE in Karaf with scripts registered using additional factory.

In particular, we created the class `GroovybasedKarafCommand` which all EKSE-based script commands should extend. It itself extends `org.apache.karaf.shell.console.OsgiCommandSupport`.

After the scripts are launched in the Karaf container using a `GroovyScriptEngine` deployed as part of the EKSE bundle, we generate a new Groovy script that lists all the script classes and returns references to them. Calling this script gives us references of all script classes in the Java context outside of Groovy, which we then forward to a class called `KarafCommandFactory`. This one in turn acts as factory for `KarafCommand` objects, which are created containing references to one script class each, and which are based on the approach of `org.apache.felix.gogo.commands.basic.SimpleCommand` extended with the functionality of providing completers by implementing `org.apache.karaf.shell.console.CompletableFunction`.

Each such object is then registered as service in Karaf by the `KarafCommandFactory` with the special properties `osgi.command.scope` and `osgi.command.function`, which are both derived from annotations in the original Groovy script files.

When Felix Gogo notices newly registered services with these specific properties, they are automatically added as new commands inside the console, which is exactly what we needed to achieve.

IV. Usage

By showing real usage examples we will now show the impact that working with EKSE has had on the integration activities at ESOC.

A. Considering EKSE as Part of EGS-CC

Due to the fact that EKSE is deployed as a part of the Karaf console, it can be deployed in various forms within EGS-CC. In particular, three different deployments can be identified:

1) EKSE can be deployed directly inside the root container of EGS-CC. This enables a system administrator to create and start new sessions using it.
2) EKSE can be deployed inside a Karaf container that belongs to an EGS-CC session. This enables an integrator to test the interaction with EGS-CC components of that particular session, e.g. to read and set parameter values.
3) EKSE can be deployed inside a completely external Karaf container. This enables the realistic development of source code snippets which are supposed to cooperate with EGS-CC remotely.

The first two of these deployments are the common way of deploying EKSE, which has already been shown in Figure 3. However, the approach of using an external Karaf container is more interesting: Here we are adding an entirely new container to the system, that would not otherwise be required any that only contains a rudimentary set

of EGS-CC bundles needed for connecting to the rest of the system. In this external container, we are then starting EKSE, as seen in Figure 9. The advantage of this approach is that we can this external container on a completely separate machine, and rely on the EGS-CC internal Infrastructure component to transmit data between the EGS-CC based system and our external container. This allows several users to have their own command lines running on separate machines with separate user sessions.

However, in the current releases of EGS-CC, the Security component is not yet integrated and therefore the advantage of such separate user sessions is not yet visible, such that for now centralized EKSE bundles are used and if a remote access is required, a simple access to the container via SSH is performed.

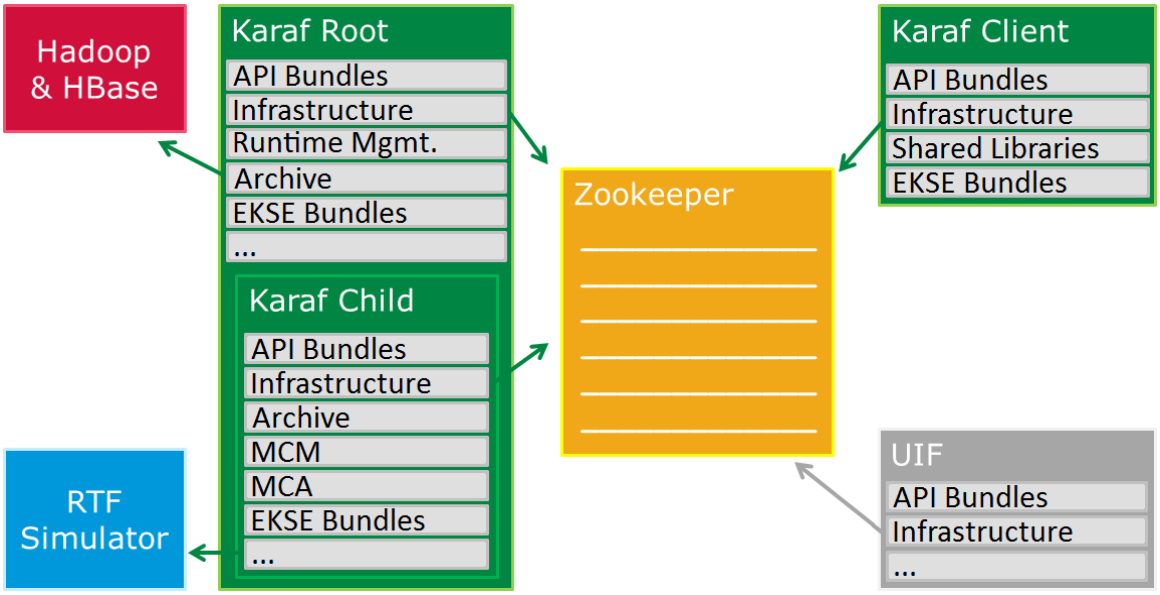

Fig. 9 EKSE deployed inside the existing root or session containers or external client container (top right).

B. Reducing Complexity by Reusing Scripts as Building Blocks

The main idea behind EKSE is to simplify the interaction with the backend of EGS-CC. The underlying OSGi APIs used by the system are very powerful and designed to be generic enough to be used in almost any monitoring and control context. The immense amount of possibilities introduced by such a generic approach ensure that EGS-CC based systems will be adaptable for even currently unforeseen needs in the future and are one of its core strengths. At the same time, all of these possibilities make the internal APIs rather complex, such that even simple tasks can be quite challenging to perform when directly accessing the backend through Java or even Groovy code.

To reduce this underlying complexity for ordinary day-to-day tasks, not just in interacting with the command line interface but even in case of creating new scripts that can be used in the future, we decided to enable the heavy reuse of previously written scripts as building blocks for new ones. Another approach could have been to disable any interoperation between any scripts, such that each script itself would be a standalone extension of EGS-CC, which would have given us more easily reusable independent parts in the future, but this would have severely slowed down the actual creation of such scripts.

C. External Automation

After the experience with the first implementation approach we considered (see III.A), we recognized the usefulness of being able to connect to an EGS-CC based system from any external context via something as simple as a shell script. However, having decided on running EKSE as part of the interactive Karaf session, we did not manage to offer this functionality to the users immediately.

To solve this problem, we focused on the underlying functionality used by Apache Karaf, which provides an SSH server as part of each container. The Karaf client through which the user interacts with EKSE internally is nothing but an SSH client connecting to this server, which forwards the commands sent by the user and returns the output from the server. This means that to send any commands, the Karaf client can be entirely circumvented and the command can be sent via SSH directly.

Having understood this, we provided a suite of shell scripts that can be used to call any EKSE command directly from the shell, or a shell script. E.g. a command to read the value of a parameter that would usually be called from inside the Karaf client as

```
egscc:readparameter mcmRoot.parameter_x
```

can equally well be called from the shell as

```
./egscc readparameter mcmRoot.parameter_x
```

D. Connecting EGS-CC with Open MCT

EKSE enables the quick modification of its underlying scripts and simple composition by which existing scripts can be leveraged to quickly create new functionality. This has been used to improve turnaround times of particular test activities and even enabled us to quickly integrate new systems into the existing EGS-CC ecosystem by exposing interfaces directly via EKSE. An example for this is the integration of an open source mission control system mainly developed by NASA, called Open MCT[6]. [5] This system has been connected to EGS-CC in less than a day by using mostly existing scripts to quickly provide a REST API compatible with Open MCT.

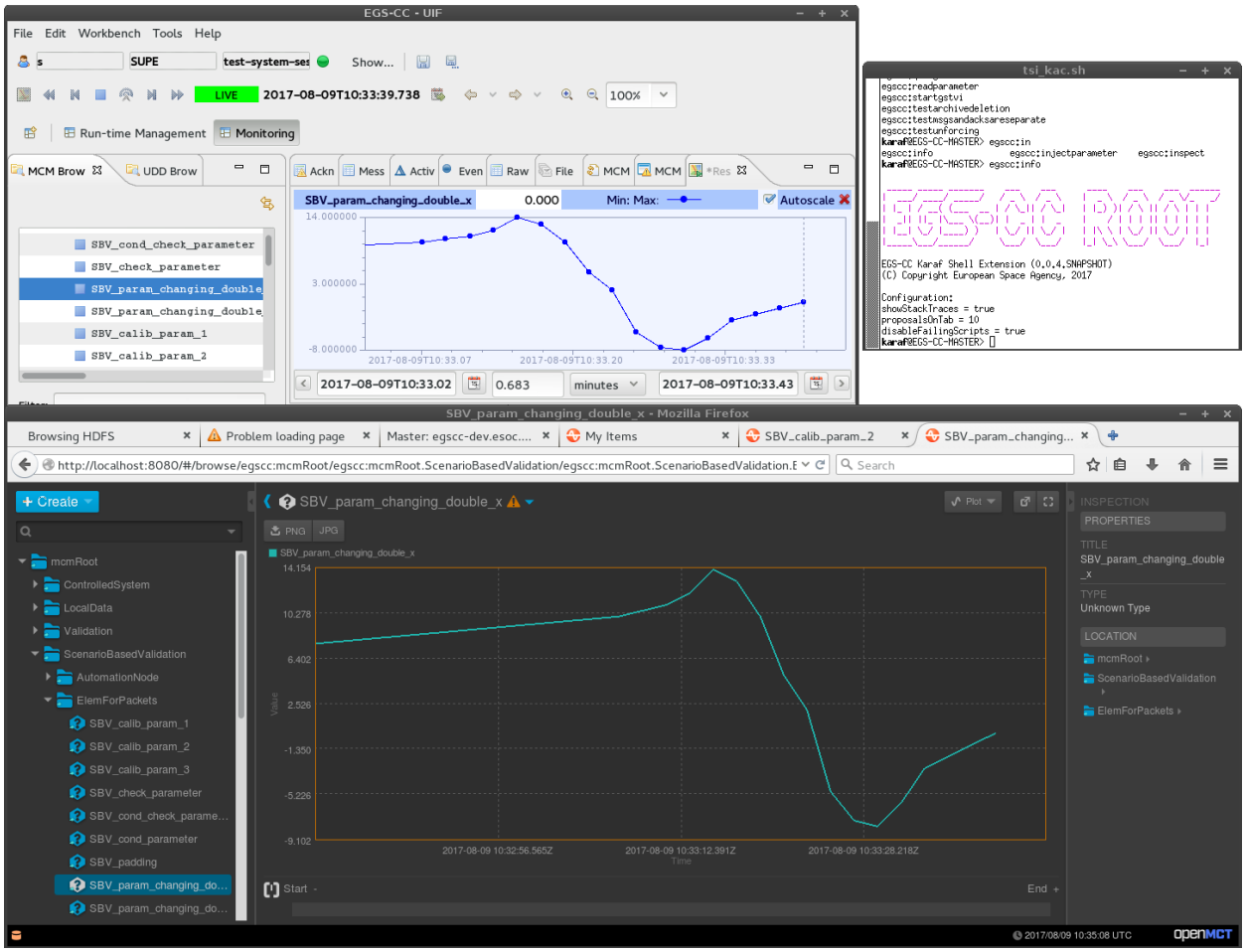

Fig. 10 EGS-CC UIF (top left) together with Open MCT (bottom), connected via EKSE (top right).

E. Test Framework

EKSE provides a test framework which can be used to write both individual tests and entire test suites, as shown in Figure 11. This framework is not part of the EKSE source code itself, but is instead purely based on Groovy scripts, such that the entire framework can be changed, if the need arises, by the user.

[6] https://nasa.github.io/openmct/

The test framework is used both to test the EGS-CC based system with which EKSE interacts, and to test the scripts run within EKSE themselves. This makes it easy to detect necessary script changes when the underlying EGS-CC APIs have changed.

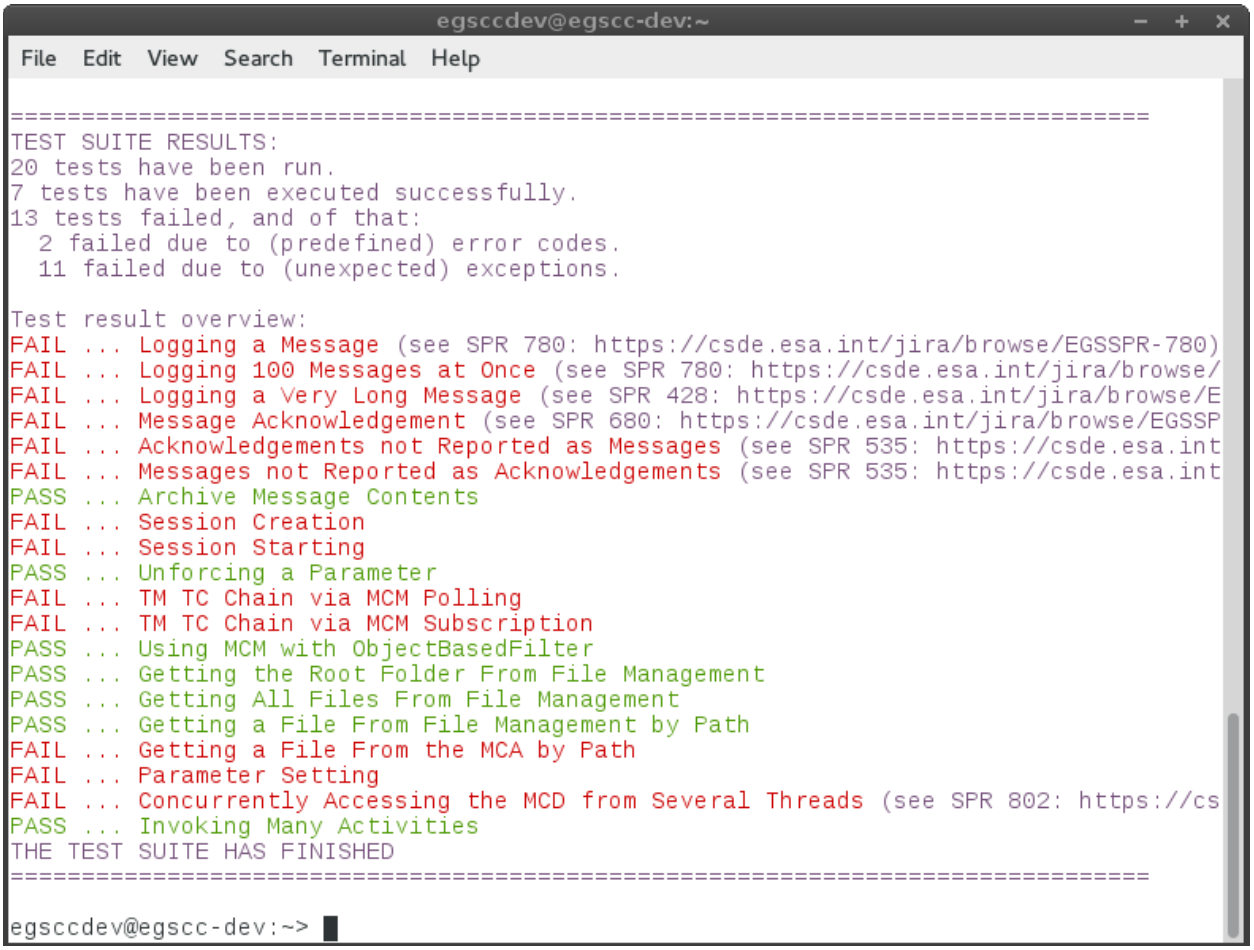

Fig. 11 Test results shown in the EKSE-based test framework.

F. Configuration Tracking Interface

EGS-CC based systems provide a configuration tracking capability, through which the interaction with different versions of the configuration of the monitored system and the interaction with different monitored systems altogether is simplified. This capability will in the future mainly be used by system administrators, who often prefer command line based interactions over graphical ones, as they can be more easily automated such that large amounts of machines can easily be configured in a similar way. Therefore, a configuration tracking interface is currently being developed at ESOC which itself is based on EKSE.

V. Challenges

Even though a lot of progress has been made and the availability of EKSE has been very helpful so far, major challenges needed to be overcome and some are still remaining.

A. Scripting Coordination

The ability to reuse existing scripts as building blocks for new ones can lead to complex dependencies between the scripts arising, such that changing one script might potentially break a large amount of scripts relying on it. Especially when the scripts which are then no longer working are not being executed soon, the failures ultimately arising when they are finally executed again can be very difficult to track down.

The test framework provided by EKSE can help with this, as after each change to the behavior or the interface of an underlying script, the test suite that tests the EKSE scripts themselves can be rerun. This immediately shows if scripts now fail to compile or throw unexpected exceptions when being run. However, even this may not find all problems, as a script may return slightly wrong data without actually throwing an exception.

The main lesson learned from this is therefore to appeal to the users to restrict the reuse of existing scripts to a sensible amount: If a complex behavior is already encapsulated in another script and is required by a new one, then this should be reused, while no dependencies between scripts should be established just for having to write one or two lines of code less. In addition, breaking interface changes especially in underlying utility scripts should be kept to a minimum, and if such changes are necessary, self tests should be used to ensure as much as possible that no existing scripts rely on the deprecated functionality.

B. Challenging Software Life Cycle

The distribution of scripts and especially the software life cycle of EKSE can be quite challenging. EKSE itself only receives a few centralized changes every couple of months, but scripts can easily change several times daily, and several users can change the same scripts for diverse purposes.

This focus on the scripts themselves can also be seen in the fact that there are currently less than 4,000 lines of Java code in the EKSE component itself, including a few inbuilt scripts, while there are over 20,000 lines of Groovy code in the script repository.[7]

Especially since every integrator can even redefine scripts for their own needs, backfeeding changes is not only logistically complicated when EKSE is being adopted in various locations, it might not even be beneficial to backfeed certain changes and some sort of authority need to oversee this process such that the overall codebase is not deteriorating in the long term.

Currently, this is tackled by coordinating contributions to scripts individually with the users involved, which however does not scale very well in case of more and more users adopting EKSE.

C. From Preliminary Test Activities to Realistically Scaled Systems

Finally, using EKSE with an operationally scaled system might lead to problems, as many of the existing scripts were originally written only for limited interactions with the EGS-CC backend during particular tests. On example is the setting of a parameter, which used to be done in such a way that a new conversation with the backend was opened every time that the script was invoked. This is perfectly fine as a self-consistent approach when only a few parameters are set over a long time, but leads to problems when realistic amounts of parameters are set during stress tests of the underlying system, as we are then no longer stress testing the backend system itself due to not even reaching the backend as we are slowed down due do our own infrastructure use.

When problems like these are detected however, they usually can easily be fixed - in the particular example of setting a parameter value, the fix only included 12 lines of source code. In general, we have made the experience that despite the fact that scripts might not always be optimized for large amounts of data and operational-scale deployments, the interaction with the system by using EKSE is generally still much more reliable than the interaction with it by using e.g. the graphical user interface, which of course is also still under active development and not fully optimized itself. Therefore, despite this potential flaw EKSE has proven to be a valuable tool numerous times by now.

VI. Conclusion

Overall, the development of EKSE has been a useful exercise and the resulting product has been used in various circumstances by different integrators and development teams of EGS-CC. The choice of reusing the existing Karaf console rather than developing an own one seems particularly well in hindsight, as most of the originally identified disadvantages of using this approach could be circumvented later on.

There currently remains ongoing work to extract useful building blocks from the core scripts defined in EKSE as basis for a toolkit to be used by various components, and the configuration tracking interface based on EKSE as of now remains to be used in a more widespread manner, but overall the development can be seen as a success.

[7] On 20th March 2018, `find . -name '*.java' | xargs wc –l` showed 3,695 lines for EKSE itself, while `find . -name '*.groovy' | xargs wc -l` showed 21,301 lines for the scripts.

Appendix A

Acronym List

EGOS-CC	ESA Ground Operation System Common Core (M&C system on top of EGS-CC)
EGS-CC	European Ground Systems Common Core
EKSE	EGS-CC Karaf Shell Extension
ESA	European Space Agency
ESCLI	EGS-CC Standalone Command Line Interface
ESOC	European Space Operations Centre
JVM	Java Virtual Machine (virtualized environment in which Java applications are executed)
M&C	Monitoring and Control
OSGi	Open Services Gateway initiative (which specifies a modular service framework for Java)
RI	Reference Implementation (of EGS-CC)
TSI	Test System Instance (integrated EGS-CC components forming an M&C system example)
UIF	Graphical User Interface delivered as part of the EGS-CC reference implementation

Appendix B

EKSE Example Script

The following example script defines a command which takes in one argument and prints it back out to the user:

```
package esa.egscc.kernel.cli.karaf.scripts;

// import Karaf-shell-related dependencies
import esa.egscc.kernel.cli.karaf.GroovybasedKarafCommand;
import org.apache.felix.gogo.commands.Argument;
import org.apache.felix.gogo.commands.Command;

// tell Karaf about the command egscc:hello
@Command(scope = "egscc", name = "hello", description="Says hello")
public class Hello extends GroovybasedKarafCommand {

   // one optional argument (for several, increase index)
   @Argument(index = 0, name = "message", description = "What should be
   said", required = false, multiValued = false)

   String message = null;

   // this method is called when the command is executed
   @Override
   protected Object doExecute() throws Exception {
      hello(message);
      return null;
   }

   // a public method performing our actual business logic
   public static void hello(String message) {
      // if the user did not enter a message...
      if (message == null) {
         // ...then take it from the HelloWorld script
```

```
            message = HelloWorld.sayHello();
        }

        // print the message to the Karaf console
        println(message);
    }
}
```

References

[1] Pecchioli, M., Carranza, J.M., Walsh, A., "The Highlights of the European Ground Systems - Common Core Initiative," *SpaceOps 2014 Conference*, SpaceOps Conferences, (AIAA 2014-1767)
doi: 10.2514/6.2014-1767

[2] Pecchioli, M., Carranza, J.M., Walsh, A., "The Operational Adoption of the EGS-CC at ESA," *SpaceOps 2016 Conference*, SpaceOps Conferences, (AIAA 2016-2304)
doi: 10.2514/6.2016-2304

[3] Plaut, J. J., et al., "Jupiter Icy Moons Explorer (Juice): Science Objectives, Mission And Instruments," *45th Lunar and Planetary Science Conference*, Texas, 2014, LPI Contribution No. 1777, p. 2717

[4] Sametinger J., Stritzinger A., "Exploratory Software Development with Class Libraries," Mittermeir R. (eds) *Shifting Paradigms in Software Engineering*, Springer, Vienna, 1992, p. 25
doi: 10.1007/978-3-7091-9258-0_4

[5] Trimble, J., "Open Source Software for Mission Operations - Technology, Licensing and Community", *SpaceOps 2014 Conference*, SpaceOps Conferences, (AIAA 2014-1762)
doi: 10.2514/6.2014-1762