The BECCAL Experiment Design and Control Software

Arnau Prat
German Aerospace Center (DLR)
Institute for Software Technology
D-38108 Braunschweig, Germany
arnau.pratisala@dlr.de

Jan Sommer
German Aerospace Center (DLR)
Institute for Software Technology
D-38108 Braunschweig, Germany
jan.sommer@dlr.de

Ayush Mani Nepal
German Aerospace Center (DLR)
Institute for Software Technology
D-38108 Braunschweig, Germany
ayush.nepal@dlr.de

Tobias Franz
German Aerospace Center (DLR)
Institute for Software Technology
D-38108 Braunschweig, Germany
tobias.franz@dlr.de

Hauke Müntinga
German Aerospace Center (DLR)
Institute for Satellite Geodesy and Inertial Sensing
D-28359 Bremen, Germany
hauke.muentinga@dlr.de

Andreas Gerndt
German Aerospace Center (DLR)
Institute for Software Technology
D-38108 Braunschweig, Germany
University of Bremen
D-28359 Bremen, Germany
andreas.gerndt@dlr.de

Daniel Lüdtke
German Aerospace Center (DLR)
Institute for Software Technology
D-38108 Braunschweig, Germany
daniel.luedtke@dlr.de

Abstract—**This paper presents the software responsible for the design and execution of the experiments in the Bose-Einstein Condensate and Cold Atom Laboratory (BECCAL) mission, an experiment with ultra-cold and condensed atoms on the International Space Station. The software consists of two parts: the experiment control software and the experiment design tools. The first corresponds to the software running on the payload and is in charge of controlling and executing the experiments, while the latter are the tools used by the scientists to create the experiment definition that will be later uploaded to the instrument to be executed. To overcome the challenge of developing software with such complexity, it was decided to follow a model-driven development approach. Several domain-specific languages (DSLs) have been created to allow scientists to describe their experiments in a domain-specific way. These descriptions are then uploaded and executed by different interpreters on-board. The paper details the architecture of the experiment control software and the different modules that compose it, as well as the developed languages and tools used to describe new experiments. The paper also discusses and evaluates some important aspects of the software, such as how resilient it is to failures, as well as the advantages and disadvantages of the selected approach compared to other approaches used in similar missions. The developed software will also be used for the MAIUS-2/3 missions.**

TABLE OF CONTENTS

Published article can be found at: https://ieeexplore.ieee.org/document/9438129

1. INTRODUCTION

The NASA-DLR Bose-Einstein Condensate and Cold Atomic Laboratory (BECCAL) [1] aims at conducting experiments with ultra-cold and condensed atoms on board the International Space Station (ISS). Its goal is to enable fundamental research as well as advance technological development by satisfying a wide range of experimental needs. The unique microgravity environment on board the ISS will contribute to these goals by providing prolonged times of free fall and observation.

The payload operation will be done in collaboration with scientist from many different institutions, and the aim is to enable all participants to design and execute their experiments on the instrument. Finding a common framework to be used by all scientists to develop new experiments for the apparatus can be a difficult task. The developed tools should be powerful enough to create any possible experiment as well as simple enough to be used by any user without in-depth knowledge of the hardware. Additionally, the tools must prevent the user from designing experiments which could harm the apparatus.

This paper presents the software tools responsible for the design and execution of the experiments in BECCAL: The experiment control software and the experiment design tools. The first one is the software running on the payload and is in charge of controlling and executing the experiments, while the latter are the tools used by the scientist to create the experiment definitions that will be later uploaded to the instrument to be executed.

Both inherit from the software [2] used in the MAIUS-1 sounding rocket [3] mission, the first to create a Bose-Einstein Condensate in space. While in the case of MAIUS-1, the microgravity phase, in which the experiments were executed, was only a few minutes long, in BECCAL this time will be much longer. This will allow for longer and more complex experiments, which the experiment control software will need to support.

The main difference of the software with respect to its predecessor is the possibility to add or modify experiment descriptions without the need of recompiling the software.

This is achieved by using interpreter engines which interpret textual experiment descriptions and execute them after successful sanity checks. This is a feature that was not needed for MAIUS-1 due to the limited scope of the mission, but for a multi-user facility such as BECCAL, this becomes mandatory.

The software implements several other features to meet the new mission requirements, such as support for new hardware compared to the MAIUS-1 mission as well as updating third party frameworks and libraries to their latest version. Also, since the experiment will be a payload on board the ISS, the overall system has to fulfill the necessary safety requirements. To this end, the design and code quality of the overall software have to meet the required standards.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of related work. Section 3 describes the experiment control software. Section 4 gives a summary of the languages used to design the experiment. Section 5 introduces the experiment design tools. Section 6 presents some results on some important aspects of the software. Finally, Section 7 states the conclusions and future work.

2. Related Work

On-board software for spaceborne experiments is usually written using languages such as C or C++. One of the reasons for this is the flexibility and performance provided by these languages [4]. However, if non-computer experts, such as in this case, need to contribute to the software, by creating new experiments, the use of these languages can become a problem due to their complexity. One solution for this are Domain Specific Languages (DSLs) [5]. Such languages are usually designed for a specific project and have the advantage of introducing only a reduced set of elements. This makes them easy to learn and use. Descriptions written using these languages can then be used to generate code to be compiled with the rest of the software or alternatively they can be executed by an interpreter.

These languages often make use of a Model-Based Software Development (MBSD) approach to specify its syntax through a precise and concrete language model. Apart from providing a high level of abstraction and making them platform independent, this approach has other advantages such as reduced manual implementation of interfaces and increase maintainability [6]. The reason for this is because the model becomes the single point of truth and redundancies can be substantially reduced.

Modeling languages such as the Unified Modelling Language (UML) and System Modelling Language (SysML) have proven to be invaluable tools for designing complex systems [7] [8]. Modeling languages can have a textual or graphical representation. For the second one, a good example is MATLAB/Simulink, which allows to graphically specify software components to later generate source code from it.

Overall, the use of such languages is a good fit for BECCAL, where non computer experts need to collaborate and the tools have to be as accessible as possible. As we will see, BECCAL implements different Domain Specific Languages (DSLs), both graphical and textual to easily create new experiment definitions. These languages are designed specifically for the project at hand, which makes them easier to use due to their reduced set of elements.

The presented approach differs from the one used in similar missions such as NASAs Cold Atom Laboratory (CAL) [9] [10]. In CAL, experiments are designed and controlled using LabVIEW taking advantage of its wide availability in industry as well as its easy to use interface. However, such approach usually requires access to a graphical session on the computer running the experiments, either via direct access or using a remote desktop sharing environment. This costs bandwidth and may be subject to delays, possibly making it difficult to operate remotely. On the other side, our approach offers similar advantages without such burden.

The described approach also differs from the one used in MAIUS-1 as mentioned in the introduction. Since experiments are not converted to C++ to be compiled together with the flight software but instead are saved as a textual file to be interpreted on-board. For MAIUS-1, compilation was feasible due to its short flight duration on a sounding rocket. However, a multi-user facility such as BECCAL requires easy uploading and change of experiments. The main advantage of this approach is that it allows adding new experiments descriptions without having to restart the software. On the other side, before interpreting them, extensive sanity checks need to be performed in case there could be errors with the experiment descriptions or input parameters. The resilience of the software due to possible errors in the experiment descriptions is evaluated in Section 6.

Another alternative used by other complex experiment control software for physical experiments is to use high-level languages such as Python, which may be seen as more accessible and simpler than C or C++. An example of this can be found in the JOKARUS [11] mission, a compact optical iodine frequency reference for a sounding rocket, in which Python was used to program the control software. However, the use of these languages usually implies a higher memory consumption and a decrease in performance compared to low-level languages, which sometimes cannot be afforded.

Another interesting example is Orocos [12], an execution environment for building real-time robotics, which allows to control robots by using a different DSL, which are later interpreted using a Lua scripting engine. This approach is very similar to ours, however the use of custom scripting languages instead of a general purpose one such as Lua allowed us to tailor the interpreters and syntax of the languages themselves to our needs and optimize it.

3. Experiment Control Software

The experiment control software is the software running on the on-board computer of the experiment. It has direct communication with the experiment electronics and its task is to execute the experiment descriptions as well as to record data from the different subsystems. The software is controlled and monitored through the ground control software. Due to the large amount of experiment hardware and domains involved, a model-driven approach was chosen for its development.

This model-driven approach is used for the hardware drivers and the experiment description part of the software. For this, a minimal core system of the on-board software provides the necessary interfaces to implement drivers and experiments. These descriptions are generated by the engineers and scientists using different DSLs. For the drivers, the generated code is compiled with the rest of the flight software since the hardware will not change during the experiment lifetime. The experiment descriptions are saved in a textual form to be later interpreted by different interpreters, which are part of the core software.

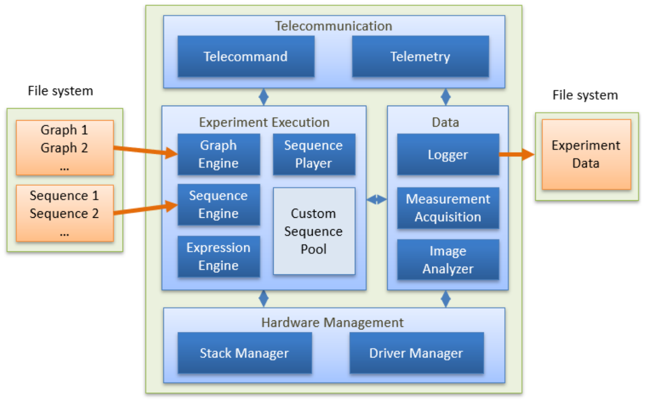


Figure 1: Experiment Control Software Architecture

Figure 1 shows the main building blocks of the software and their inter-dependencies. Starting from the bottom, we have the controllers for the experiment hardware. We can distinguish between two types: The Stack Manager handles custom-built hardware using drivers auto-generated from hardware descriptions provided by the engineers. The Driver Manager handles off-the-shelf hardware such as cameras via drivers manually coded. On top we have the different Interpreter Engines, which interpret and execute the experiments descriptions written by the scientists as well as the module in charge of scheduling the execution of the experiments. Finally, both above and to the right we have the core modules of the software, which performs tasks such as logging as well as command and data handling.

Communication between the different modules is realized by the "Tasking Framework" developed by DLR [13] which has been successfully used in the MAIUS-1 mission and other DLR projects [14]. In this framework, modules are connected through so-called channels (represented as arrows in Figure 1). These are unidirectional and handle a certain datatype that has to fit the input and output slots of the connected modules. In BECCAL these datatypes are called packets. Tasking Framework follows an event-driven approach, which means that when a module receives a new packet a response procedure will be triggered.

These modules can be gathered into different groups depending on their functionality. Namely four groups have been identified: Telecommunication, Experiment Execution, Hardware Management, and Data Management. Next, each of these groups is described.

The Telecommunication group handles all communication that is exchanged live with the Ground Control Software. It is in charge of accepting telecommands from ground and sending back telemetry packets. The received telecommands are checked for validity and forwarded to the responsible module for further processing. Telemetry data from other software modules is packaged into telemetry packets according to the communication protocol and sent to ground. It is also the responsibility of the Telecommunication group to monitor the state of the ground connection and reestablish it in case the connection is lost.

The Experiment Execution group is separated into two levels: sequences and graphs. Sequences are essentially building blocks of the experiment and are used to carry out actions on the experiment hardware. The graph chains multiple sequences into a full experiment cycle and even allow for a certain amount of flow control depending on measured conditions. Both the sequences as well as the graphs are saved as files and are loaded by the Experiment Execution group. The Experiment Execution group contains all software parts necessary to ensure the correct execution order of the experiment. This includes the interpreter for the experiment sequences and graphs as well as the control flow defined therein. The group then passes the actions to the Hardware Management group for execution.

The Hardware Management group is the only software that directly access the experiment hardware. It is responsible for controlling the stack electronics as well as the cameras and other hardware operations. Additionally, to the control of the experiment hardware, this module also collects raw data from the electronics and the rest of the hardware and send it to the Data Management group.

The Data Management group processes all raw data that is collected from the hardware. This group does not have direct access to the hardware and it receives the raw data from the Hardware Management group. The group converts the incoming raw data into the respective physical units. Additionally, to the conversion of raw numeric data, the Data Management module is also responsible for downscaling images for the live telemetry downlink.

The processed data is then distributed threefold: It is partly fed back to the Experiment Execution group to determine the experiment flow. Parts are sent to the Telecommunication group and then processed as live telemetry data for the ground station. This also includes low resolution versions of the acquired images. Finally, the full set of data, including full resolution images, is saved to the file system and can be downloaded during idle periods of the experiment.

A special group, which does not appear in Figure 1. It is the Watchdog group, which is not part of the experiment control system but of the Failure Recovery System. The main purpose of the module is to monitor the remaining groups and report and restart them if one of it is failing. To this end, this uses a Failure Detection Isolation and Recovery (FDIR) strategy.

The on-board computer running the experiment control software uses a standard Linux operating system. Uploading the experiment files and download of scientific data can be done using standard file transfer protocols like Secure File Transfer Protocol (SFTP). The system also allows carrying out maintenance such as software update through direct access to the computer using protocols such as SSH.

4. Experiment Languages

This section goes into more details regarding the different languages used to describe experiments. In total five different languages have been developed, which can be split into two domains: electronic domain and experiment domain. The languages corresponding to the electronics domain are used to provide hardware definitions that will be used to generate source code. Whereas the languages corresponding to the experiment domain are used to describe the experiments to be executed on the hardware and are the languages that this paper will focus on. This domain is formed by three DSLs. First, is the Sequence DSL, which is an abstract description of the behavior of a single step of the experiment. Additionally, to prevent repeating code, Sequences are subdivided into Subsequences as many parts of sequences will be repeated and only differ by parameters or timing. This corresponds to the Subsequence DSL. The last layer of the model is a graphical representation of the experiment flow called Experiment Execution Graph, which is designed as a binary decision graph. This can be seen in Figure 2.

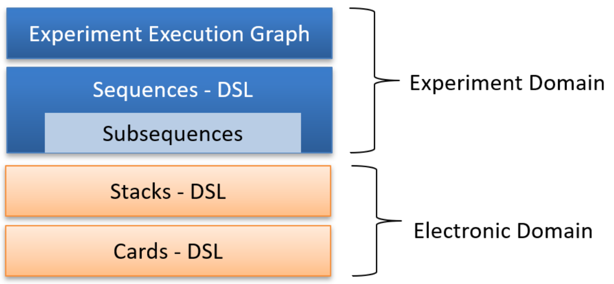


Figure 2: Developed Domain Specific Languages

The syntax of the Sequences, Subsequences and generated experiment execution graphs is based on YAML (YAML Ain't Markup Language). The main reasons why it was chosen was because it was found to have a good trade-off between human and machine readability. Although it is possible to edit these files manually, it is intended that the creation of new experiments is done through the experiment design tools presented in Section 5. These tools provide verification and validation, warning the user of possible errors, and simplify the overall process of creating new experiments.

Communication of the sequences with the electronics is done through the so-called channels and they are the basic elements to interact with the experiment hardware. There are two types of channels: output channels, which change some physical parameter (actuators) and input channels, which measure a certain parameter (sensors). In a sense, sequences and subsequences can be seen as a precise timing description of the state of the apparatus, where each step describes which and how a channel is modified. On the other side, graphs can use input channels to read the current state of the apparatus and decide what next step to take.

In Figure 3 we can see an example of a subsequence file. The structure of sequences and subsequences is fairly similar and a brief description of the structure for both is given below.

A sequence can contain parameters, constant channels and subsequence elements. Each of them is described below. Also, a sequence should have a unique name and identifier. Optionally, it is possible to specify a description for the sequence and the date when it was created or modified. After these, one can specify the parameters that the sequence takes. The value that each parameter takes can then be specified when running the sequence. If no value is specified or the value is not within defined bounds, the default value is used. After the parameters, one can specify constant channels. These are channels that remain constant during the execution of the sequence. Its value can be of three types: boolean, float or ramp. Finally, one can specify subsequences. In order to define which subsequences should be executed, first

```
name: TestSubseq
description: 'An example test subsequence'
date: '01.01.2021'
init: false
parameters:
  Analog0: {default: 0.0, max: 10.0, min: -10.0}
  Analog1: {default: 0.0, max: 10.0, min: -10.0}
  Time: {default: 10.0, max: 500.0, min: 0.0}
subsequence:
- time: 0.0
  slotname: SetToZeroAnalog123
  channels:
  - {name: AnalogOut00, value: 0.0}
  - {name: AnalogOut01, value: 0.0}
  - {name: AnalogOut02, value: 0.0}
- time: 0.1 + Time
  slotname: SetValuesAnalog12
  channels:
  - {name: AnalogOut00, value: Analog1}
  - {name: AnalogOut01, value: Analog2}
```

Figure 3: Example Subsequence File

it needs to be specified the time when they should be executed followed by the subsequence name. Note that the time should be greater than the previous subsequence time plus the time it takes for such subsequence to execute.

Similar to a sequence, a subsequence may contain zero or more parameters, and slots elements. Each of them is described below. Also, each subsequence should have a unique name. In the same way as the sequence, a subsequence starts with its metadata fields, i.e., its unique name and optionally description and date. The init option is used to specify if the channels will be called in init mode. While sequences are comprised of subsequences, subsequences are comprised of slots. Each slot allows one to specify a set of channels that needs to be called at a certain period of time. The value for these channels can be a parameter, float or bool.

As seen in Section 2, the use of DSLs has several advantages. However, it also limits what is possible to do with them. Thus, there are certain sequences that are not realizable using this syntax and had to be coded directly using C++. These sequences are called Untimed Sequences and are compiled together with the flight software. The normal sequences are known as Timed Sequences. An example of Untimed Sequences is the sequence in charge of locking the lasers to the right frequency.

It is foreseen that a pool of commonly used sequences and subsequences (both timed and untimed) will be shared with the experiment developers containing basic functionalities such as taking a picture. Then this pool can be used as basic blocks to create more complex experiments using them as basic elements for the graphs. While additional timed sequences will be able to be created using the experiment editors, new untimed sequences will need to be coded by the payload developers.

The graphs on the other side are graphical representations designed as a binary decision graph. Similar to UML activity diagrams, graphs are represented by boxes and decision points. Each decision point is shaped like a rhombus and has a binary output (true or false). The output will depend on the evaluation of the expression inside the decision point. Graphs can have different types of boxes, these are: assignment boxes, where an assignment to a parameter can be done; sequence boxes, which can call a given sequence; subgraph boxes, which can call another graph and scan-fit blocks, which allow repeating a sequence changing the value

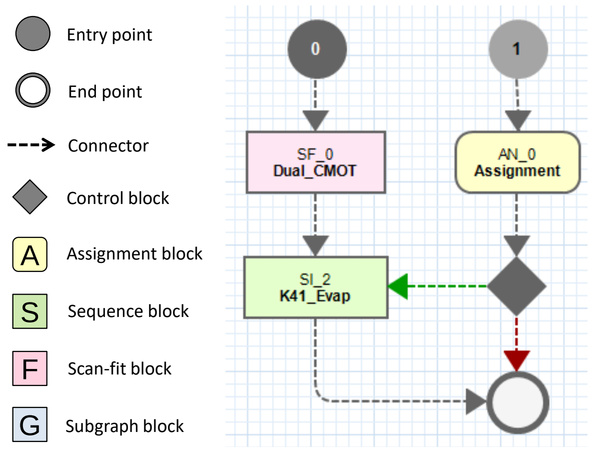


Figure 4: An Example Experiment Execution Graph

of different parameters that the sequence takes. Depending on the type of box it will have a different color or shape. Elements are connected together through arrows and each element can only be connected to another element. Figure 4 visualizes an example sequence-graph with two entry-points. A graph can be started from either of these entry-points, and a graph execution terminates at an end-point.

Apart from sequences and control blocks, the graphs can also include other special components. One of the most important are the so-called scan-fit blocks. These allow to scan a sequence with different values for certain parameters and optimize them by analyzing the output of the sequence being run. These have been substantially improved with respect to the previous version flown in MAIUS-1 allowing more complex scans and optimization of multiple parameters simultaneously.

```yaml
name: GraphTest_conf1
id: 0x22451BE4
description: 'An example test graph'
date: '02/09/2020 10:36:30'
globals:
  glob1: {default: 1.0, max: 2.0, min: -1.0}
  glob2: {default: -1.0, max: 2.0, min: -1.0}
points:
- {id: 0x00000000, next: 0x00000101}
- {id: 0xFFFFFFFF, next: 0xFFFFFFFF}
- {id: 0x00000001, next: 0x00000102}
controls:
- {id: 0x00000104,
   condition: 'glob1 < measurement(NTC_2DCoil)',
   nextTrue: 0x00000105,
   nextFalse: 0xFFFFFFFF}
- {id: 0x00000102,
   condition: 'glob2 := 2.0',
   nextTrue: 0x00000104,
   nextFalse: 0x00000104}
sequences:
- {id: 0x00000105,
   name: SI_2,
   hash: 0x9F50D140,
   parameters: [1000.0, 1.0, 1.0, 800.0, 800.0],
   next: 0xFFFFFFFF}
subgraphs:
- {id: 0x00000101,
   name: scanfits/GraphTest_conf1_SF_0_0x09AC94EA,
   next: 0x00000105}
```

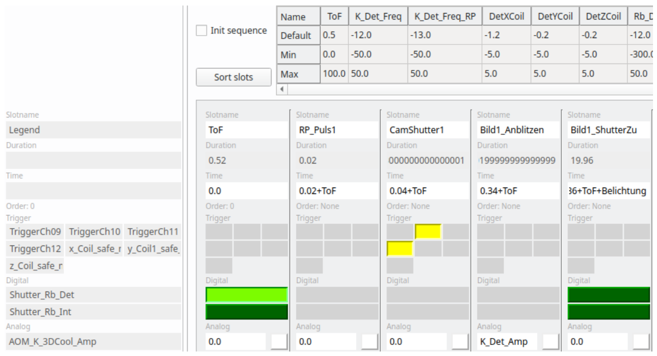
Figure 5: A Generated Flow File Example

In order to be interpreted, graphs are translated into so called flow files. Similar to sequences, flow files are also based on YAML and they are an element representation of graphs. Figure 5 shows an example of one of these files. All the elements are grouped based on these basic types. In this case, entry points, sequences, subgraphs and control blocks. Assignment blocks are translated into control blocks, and scan-fits are generated into subgraphs in which basic elements emulates the behavior.

5. Experiment Design Tools

The experiment design tools are used by the scientist to design new experiments. They support the scientist with designing a formally correct experiment which later can be uploaded and executed on the payload. There are two main tools: the Sequence GUI and the Experiment Editor. Both software packages provide graphical user interfaces to assists physicists to design complex experiments. The Sequence GUI has been used to design the Sequences and the Subsequences, whereas the Experiment Editor is used to design the experiment execution graphs. In addition, the Sequence GUI can also be used to locally control the experiments by directly uploading sequences to the apparatus. Both tools are foreseen to be delivered to the scientists as a single package. A more detailed description of both tools is given below.

Figure 6 shows a screenshot of the Sequence GUI, where values for different channels of a subsequence are displayed with GUI controls inside slots, allowing for easy editing of subsequences. Triggers and digital channels are displayed as buttons, while inputs for analog channels allow floating point numbers with optional parameters. On the top, subsequence parameters can be added, removed or changed.



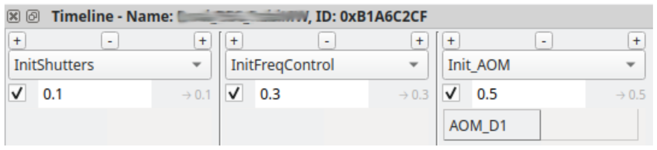
Name	ToF	K_Det_Freq	K_Det_Freq_RP	DetXCoil	DetYCoil	DetZCoil	Rb_D
Default	0.5	-12.0	-13.0	-1.2	-0.2	-0.2	-12.0
Min	0.0	-50.0	-50.0	-5.0	-5.0	-5.0	-300.(
Max	100.0	50.0	50.0	5.0	5.0	5.0	50.0

Slotname	Legend	ToF	RP_Puls1	CamShutter1	Bild1_Anblitzen	Bild1_ShutterZu
Duration		0.52	0.02	000000000000001	199999999999999	19.96
Time		0.0	0.02+ToF	0.04+ToF	0.34+ToF	36+ToF+Belichtung
Order	0	0	None	None	None	None
Trigger	TriggerCh09 TriggerCh10 TriggerCh11 TriggerCh12 x_Coil_safe_r y_Coil1_safe_ z_Coil_safe_r					
Digital	Shutter_Rb_Det Shutter_Rb_Int					
Analog	AOM_K_3DCool_Amp	0.0	0.0	0.0	K_Det_Amp	0.0

Figure 6: Screenshot Sequence GUI

An additional GUI element shown in Figure 7 allows arranging and parameterizing subsequences into sequences.

The Sequence GUI is intended for the design of sequences and subsequences, while experiment execution graphs are designed using the Experiment Editor explained below. Since sequences can only write to analog and digital items but not read from them, they cannot react to the current state of the experiment. This behavior is achieved through the experiment execution graphs by passing different parameters to the sequences they call depending on the current execution state.


Timeline - Name: [redacted], ID: 0xB1A6C2CF

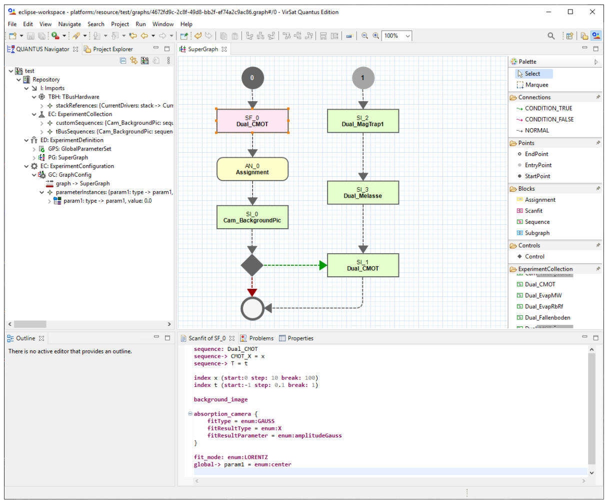
InitShutters	InitFreqControl	Init_AOM
✓ 0.1 →0.1	✓ 0.3 →0.3	✓ 0.5 →0.5
		AOM_D1

Figure 7: Sequence GUI Timeline

Input errors such as invalid values for physical channels or overlapping subsequences are checked at runtime and immediate feedback is given to the user with details about the errors. These checks are powered by the same hardware definitions that are used by the experiment control software. Therefore the hardware definitions are used as the single point of truth throughout all software tools.

The Sequence GUI is written in Python with Qt bindings for the GUI elements, which allows for high-level graphics support as well as easy debugging and extensibility by scientists.

The experiment editor is based on Java/Eclipse and provides installable features with plugins. This modular approach allows to have an incremental development workflow, so that features could be added or updated conveniently as per the new requirements from the scientists. The experiment editor follows a model-driven development methodology and is built on top of Virtual Satellite 4 (VirSat4) [15], an open source software for model-based systems engineering (MBSE). The MBSE approach in the development increases productivity by allowing source code, test files, as well as documentations to be generated automatically from the data model of the system. To that end, the experiment editor provides textual as well as graphical DSLs to describe and configure the model i.e. the experiment execution graph. Moreover, through the import mechanism sequences and hardware DSLs can be linked to the model. New textual DSLs have been developed to define condition expressions, assignment expressions, and the scan-fit operation.



```
sequence: Dual_CMOT
sequence-> CMOT_X = x
sequence-> T = t

index x (start:0 step: 10 break: 100)
index t (start:-1 step: 0.1 break: 1)

background_image

absorption_camera {
    fitType = enum:GAUSS
    fitResultType = enum:X
    fitResultParameter = enum:amplitudeGauss
}

fit_mode: enum:LORENTZ
global-> param1 = enum:center
```

Figure 8: Screenshot Experiment Editor GUI

Figure 8 shows a screenshot of the experiment editor GUI, where an example experiment execution graph can be seen. An example scan-fit operation using the respective DSL can be seen in the bottom panel. The panel on the left shows the project tree and the one on the right shows the palette. The user can choose elements from the palette and click anywhere on the graph to create it and connect them using the connectors. All graphs start at an entry point and end at the endpoint. Each of these blocks can be customized individually by opening them in an editor. A double-click on the DSL blocks (assignment, control, and scan-fit) opens the respective DSL textual editor, whereas, for other blocks the respective editor GUI is open. A graph instance can be configured by creating a graph configuration for it. A graph configuration holds an instance of all global parameters associated with the graph. Initial values of the global parameters can be assigned in the graph configuration. The flow files are generated from the graph configuration triggering a generator. A flow file is generated for the graph and all other graphs which are subgraphed by the configured experiment execution graph. Furthermore, all scan-fit operations are treated as a subgraph and they are also serialized into individual flow files.

6. Results

The BECCAL instrument is scheduled to be launched and integrated on the ISS in 2024 and planned to be operated for several years. In order to support such long lead times, the experiment control software as well as the experiment design tools need to be developed with long time maintainability in mind. For the experiment control software, this was comparably easy to achieve. It mostly depends on a standard C++ compiler and libraries provided by the operating system, which will be available given common support schedules of commercial Linux operating systems.

For the experiment design tools, this was more difficult to achieve since GUI components are often subject to constant progress and changes. We therefore use Virtual Satellite as a base-platform for our new tools, which is developed by DLR. It is a strategic software product for DLR's model-based systems engineering effort which ensures long-term maintenance. The GUI elements for the Experiment Editor to create and manipulate graphs had to be re-implemented based on Graphiti, a more modern and future-proof framework than the Eclipse Modeling Framework (EMF) used in MAIUS-1. Similarly, the Sequence GUI is updated to the most recent version of Python and corresponding Qt bindings. In both cases, care was taken to keep continuity in the user interface.

A first version of the new BECCAL software, both experiment control software and experiment design tools, have been disseminated to scientists working with a laboratory setup. The received feedback from the experts who use the tools through different interviews is very positive.

Although the user interface of the new experiment design tools were re-implemented in large parts compared to MAIUS-1, only little time was necessary to train the users to work with the new user interface. It is now already part of the daily work in the laboratory. The round-trip time for experiment changes has been decreased significantly. Previously, when changes were introduced to sequences or experiment execution graphs, the experiment control software needed to be recompiled, which could take up to 20 min in worst-case scenarios. Recompilation is now only necessary if changes were made to the control electronics of the experiment which occurs very rarely in the laboratory and will not occur at all for the final BECCAL instrument. Given the significant increase in complexity of BECCAL compared to earlier instruments like MAIUS-1, also longer and more complex test campaigns in the laboratories are to be expected.

The achieved time savings in the daily work for the scientists will facilitate this work greatly.

BECCAL is developed with an international collaboration of scientists for the experiment design in mind. Being able to exchange knowledge and experience as well as trace changes and contributions is therefore necessary. With the new BECCAL software, all experiment input data, i.e., sequences, subsequences, and experiment execution graphs are now available as simple text files in YAML format. That means, common tools for distribution, version control, and data comparison known from the software development domain can be used to establish the framework for the collaboration. The YAML format also ensures a certain level of human readability of all input data, giving the chance for manual check, if necessary.

A big change with respect to the MAIUS-1 software is the fact of using custom interpreters for the DSLs. An interesting metric is to see how resilient is our software to corrupted or invalid experiments descriptions as well as invalid input values to the experiment. In MAIUS-1, if there was a problem with an experiment definition, this could be notified by the compiler. However, in BECCAL sanity checks and validation have to be performed on the fly.

In the case of BECCAL, we can analyze three scenarios. A first one in which an experiment definition files (sequence, subsequence or graph) is corrupted and cannot be processed; a second one, in which an out-of-bounds value for a parameter is passed to an experiment; and finally, the case in which a graph goes into an error state due to the dynamics of the experiment outputting wrong values or entering into an infinite loop.

The first scenario is easily solved by using a checksum in the experiment definitions. The experiment control software computes the hash for the file and checks whether it matches the provided one. In case the hashes do not match, the definition is deemed corrupted, the experiment is not executed and a warning is sent to the experiment operator.

The second scenario is handled on the fly through validators implemented in the interpreters. When assigning a value to a parameter, the experiment control software first checks whether the value is between the defined maximum and minimum bounds. If yes, the value is assigned to the parameter. And if not, the default value is assigned and a warning message is sent to the operator.

The third case can only be checked through simulation, either by running it in a simulator or in one of the planned ground test beds. Future improvement could include to perform a model checking analysis on the graph. Every experiment needs to be tested before being uploaded to the ISS, since there are rare combinations of sequences which could have the potential to create strong heating and potentially degrading the experiment performance.

For this reason, experiments will only be allowed to run on the ISS if they have been tested and qualified by one of the ground test-beds. For qualification, an operational procedure is in place which ensures that potentially damaging experiment configurations will not be allowed to be transmitted to the BECCAL instrument on-board the ISS.

7. Conclusion

In this paper, we presented the software responsible for the design and execution of the experiments in the BECCAL mission. This is composed of two parts: the experiment control software, which is the software running on the on-board computer of the apparatus and is in charge of executing the experiments, and the experiment design tools, which are the tools used by the scientist to design new experiments. Both inherit from the MAIUS-1 software and the main novelty is the possibility to add and execute new experiments on the fly, this is done by using an interpreter engine which interprets and executes the experiment definitions without the need of recompile and restarting the software. At the moment BECCAL, is expected to fly in 2024. And several improvements will still be made to the software. However, part of the software will already be tested for the MAIUS-2/3 missions, which are expected to fly in 2022 and 2023 respectively.

Acknowledgments

This work is partly supported by the German Space Agency (DLR) with funds provided by the Federal Ministry for Economic Affairs and Energy (BMWi) due to an enactment of the German Bundestag under Grant Nos. 50WM1131-1137, DLR50WP1431-1435, 50WM1552-1557, 50WP1700-1706 and 50WM1952-1957. We would also like to thank the BECCAL and MAIUS teams for their contribution and support.

References

[1] K. Frye, S. Abend, W. Bartosch, A. Bawamia, D. Becker, H. Blume, C. Braxmaier, S.-W. Chiow, M. A. Efremov, W. Ertmer, P. Fierlinger, N. Gaaloul, J. Grosse, C. Grzeschik, O. Hellmig, V. A. Henderson, W. Herr, U. Israelsson, J. Kohel, M. Krutzik, C. Kürbis, C. Lämmerzahl, M. List, D. Lüdtke, N. Lundblad, J. P. Marburger, M. Meister, M. Mihm, H. Müller, H. Müntinga, T. Oberschulte, A. Papakonstantinou, J. Perovšek, A. Peters, A. Prat, E. M. Rasel, A. Roura, W. P. Schleich, C. Schubert, S. T. Seidel, J. Sommer, C. Spindeldreier, D. Stamper-Kurn, B. K. Stuhl, M. Warner, T. Wendrich, A. Wenzlawski, A. Wicht, P. Windpassinger, N. Yu, and L. Wörner, "The Bose-Einstein Condensate and Cold Atom Laboratory," *arXiv*, dec 2019. [Online]. Available: http://arxiv.org/abs/1912.04849

[2] B. Weps, D. Lüdtke, T. Franz, O. Maibaum, T. Wendrich, H. Müntinga, and A. Gerndt, "A model-driven software architecture for ultra-cold gas experiments in space," in *Proc. of the International Astronautical Congress, IAC*, 2018.

[3] D. Becker, M. D. Lachmann, S. T. Seidel, H. Ahlers, A. N. Dinkelaker, J. Grosse, O. Hellmig, H. Müntinga, V. Schkolnik, T. Wendrich, A. Wenzlawski, B. Weps, R. Corgier, T. Franz, N. Gaaloul, W. Herr, D. Lüdtke, M. Popp, S. Amri, H. Duncker, M. Erbe, A. Kohfeldt, A. Kubelka-Lange, C. Braxmaier, E. Charron, W. Ertmer, M. Krutzik, C. Lämmerzahl, A. Peters, W. P. Schleich, K. Sengstock, R. Walser, A. Wicht, P. Windpassinger, and E. M. Rasel, "Space-borne Bose-Einstein condensation for precision interferometry," *Nature*, vol. 562, no. 7727, pp. 391–395, 2018. [Online]. Available: https://doi.org/10.1038/s41586-

[4] M. Nahas and A. Maaita, "Choosing appropriate programming language to implement software for real-time resource-constrained embedded systems," *Embedded Systems-Theory and Design Methodology*, 2012.

[5] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[6] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[7] B. Selic, "Using UML for modeling complex real-time systems," in *Languages, compilers, and tools for embedded systems*. Springer, 1998, pp. 250–260.

[8] P. David, V. Idasiak, and F. Kratz, "Reliability study of complex physical systems using sysml," *Reliability Engineering & System Safety*, vol. 95, no. 4, pp. 431–450, 2010.

[9] D. C. Aveline, J. R. Williams, E. R. Elliott, C. Dutenhoffer, J. R. Kellogg, J. M. Kohel, N. E. Lay, K. Oudrhiri, R. F. Shotwell, N. Yu *et al.*, "Observation of bose–einstein condensates in an earth-orbiting research lab," *Nature*, vol. 582, no. 7811, pp. 193–197, 2020.

[10] M. Soriano, D. Aveline, M. Mckee, K. Virkler, C. Yamamoto, and A. Sengupta, "Cold atom laboratory mission system design," pp. 1–11, 2014.

[11] K. Döringshoff, F. B. Gutsch, V. Schkolnik, C. Kürbis, M. Oswald, B. Pröbster, E. V. Kovalchuk, A. Bawamia, R. Smol, T. Schuldt *et al.*, "Iodine frequency reference on a sounding rocket," *Physical Review Applied*, vol. 11, no. 5, p. 054068, 2019.

[12] M. Klotzbücher, P. Soetens, and H. Bruyninckx, "Orocos rtt-lua: an execution environment for building real-time robotic domain specific languages," in *International Workshop on Dynamic languages for RObotic and Sensors*, vol. 8, 2010.

[13] Z. A. Haj Hammadeh, T. Franz, O. Maibaum, A. Gerndt, and D. Lüdtke, "Event-driven multithreading execution platform for real-time on-board software systems," in *Proceedings of the 15th annual workshop on Operating Systems Platforms for Embedded Real-time Applications*, 2019, pp. 29–34.

[14] C. J. Treudler, H. Benninghoff, K. Borchers, B. Brunner, J. Cremer, M. Dumke, T. Gärtner, K. J. Höflinger, D. Lüdtke, T. Peng *et al.*, "ScOSA-scalable on-board computing for space avionics," in *Proceedings of the International Astronautical Congress, IAC*, 2018.

[15] SC-SRV DLR. (2019) Virtual satellite 4 - core. [Online]. Available: https://github.com/virtualsatellite/VirtualSatellite4-Core

Biography

Arnau Prat *received his B.S. degree in electronics systems engineering and M.S. degree in telecommunications engineering from the Polytechnic University of Catalonia (UPC), Barcelona, Spain in 2015 and 2017 respectively. He is currently a research scientist at the German Aerospace Center (DLR) in the department of Software for Space Systems and Interactive Visualization since 2018 where he is involved in the development of on-board software for space mission. From 2016 to 2017, he was a research assistant with the department of Signal Theory and Communications, UPC, Barcelona, Spain, where he was involved in a terahertz radar system. In 2017 he was a visiting student with the department of Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, USA, where he worked on an intelligent cognitive assistant for space applications.*

Jan Sommer *received his M.Sc. in Space Science and Engineering in 2013 Technical University of Luleå. After a traineeship with the European Space Agency in the section for software engineering he is now with the German Aerospace Center (DLR) in the department "Software for Space Systems and Interactive Visualization" since 2015 where he is active in the development of on-board software for spacecraft missions. His current research interests include the application of model-driven software development methods for on-board software development.*

Ayush Mani Nepal *received his M.Sc. degree in Computational Sciences in Engineering (CSE) with a major in Electrical Engineering from the Technical University of Braunschweig in 2019. He joined the department of Software for Space Systems and Interactive Visualization at the German Aerospace Center (DLR) during his Masters in year 2017 and has since been active in the development of model-driven software engineering tools for space systems. After writing the Master thesis in 2019, he is working as a scientific researcher in the same department at DLR. His main research interests include machine learning for space domain applications.*

Tobias Franz *received his M.Sc degree in Computer Science from the Technical University of Braunschweig in 2018. He joined the Institute for Software Technology at German Aerospace Center (DLR) in 2012 as part of a university program, where he was active in area of model-driven software development for embedded systems. Currently he is a research scientists with interests in model-based systems engineering for space systems.*

Hauke Müntinga *received his diploma in Physics from the University of Oldenburg in 2008. He then joined the Center of Applied Space Technology and Microgravity at the University of Bremen, where he worked on quantum optical experiments in microgravity in drop-tower and sounding-rocket experiments. In 2019, he reveived his doctorate in Experimental Physics. In 2020, he joined the Institute for Satellite Geodesy and Inertial Sensing at the German Aerospace Center (DLR), where he develops experiment control software for spaceborne experiments and simulations of quantum sensors.*

Andreas Gerndt *is the head of the department "Software for Space Systems and Interactive Visualization" at the German Aerospace Center (DLR). He received his degree in computer science from Technical University, Darmstadt, Germany in 1993. In the position of a research scientist, he also worked at the Fraunhofer Institute for Computer Graphics (IGD) in Germany. Thereafter, he was a software engineer for several companies with focus on Software Engineering and Computer Graphics. In 1999 he continued his studies in Virtual Reality and Scientific Visualization at RWTH Aachen University, Germany, where he received his doctoral degree in computer science. After two years of interdisciplinary research activities as a postdoctoral fellow at the University of Louisiana, Lafayette, USA, he returned to Germany in 2008 to work for DLR in the domain of aerospace software research. Since 2019, he is also Professor in High-Performance Visualization at University of Bremen, Germany.*

Daniel Lüdtke *received the diploma degree Dipl.-Ing. in Computer Engineering from Technische Universität Berline (Germany) in 2003. He worked as a research assistant at the department of Computer Engineering and Microelectronics, TU Berlin. He joined the German Aerospace Center (DLR), Institute for Software Technology in 2010. Since 2012 he is managing the research group Onboard Software Systems and is vice head of the department Software for Space Systems and Interactive Visualization. His current research interests include model-driven software engineering for space systems with an emphasis on reconfigurable embedded systems.*