IAC-18-A2.5.6

A Model-driven Software Architecture for Ultra-cold Gas Experiments in Space

Benjamin Weps[a,b], Daniel Lüdtke[a]*, Tobias Franz[a], Olaf Maibaum[a], Thijs Wendrich[c], Hauke Müntinga[d], Andreas Gerndt[a]

[a] *Simulation and Software Technology, German Aerospace Center (DLR), Lilienthalplatz 7, 38108 Braunschweig, Germany*
[b] now at *Jena Optronik GmbH, Otto-Eppenstein-Straße 3, 07745 Jena, Germany*
[c] *Institute of Quantum Optics and QUEST-Leibniz Research School, Leibniz University Hannover, Welfengarten 1, 30167 Hannover, Germany*
[d] *Center of Applied Space Technology and Microgravity (ZARM), University of Bremen, Am Fallturm 2, 28359 Bremen, Germany*
* Corresponding Author: daniel.luedtke@dlr.de

Abstract

Developing software for large and complex experiments is a challenging task. It must incorporate many requirements from different domains, all with their own conceptions about the overall systems. An additional level of complexity is added if the experiment is conducted autonomously during a sounding rocket flight. Without a proper software architecture and development techniques, achieving and maintaining a high code quality is a very cumbersome task.

This paper describes the architecture and the model-driven development approach we used to implement the control software of the experiments in the MAIUS-1 mission (matter-wave interferometry in microgravity). In this mission, the software had to handle around 150 experiments in six minutes autonomously and adapt to changes in the control flow according to real-time data from the experiment.

The MAIUS-1 mission was the first mission to create Bose-Einstein condensates in space and conduct other experiments with ultra-cold gases on a sounding rocket. Besides the scientific goals in the area of quantum-optics, other important objectives of the mission were the miniaturization and further development of laser systems, vacuum components, optical sensors, and other related technologies. To fulfil these goals, new experimental hardware has been created which had to be integrated and tested with the software of the experiment computer.

The custom-made hardware and the considerable number of domains involved brought up many challenges for the software engineering. To face all these challenges of developing software with this high complexity, we chose to follow a model-driven software development approach. Several domain-specific languages (DSLs) accompanied with specialized tools were created to allow the physicists and electronic engineers to describe system components and the experiments in a domain-specific way. These descriptions were then automatically transformed in C++ code for the flight software. This way we could actively incorporate all the domains involved in conducting the experiment directly in building the flight software without compromising the software quality.

We created a versatile software platform not only for the MAIUS-1 mission but also for upcoming missions with similar experiments and hardware. With our approach we were able to generate around 84% of the source code for the final flight software from the domain-specific models. Besides the improvement of the development process, the code generation made a significant contribution to the overall software quality as almost all manual coding of error-prone boilerplate code could be mitigated.

Keywords: software engineering, model-driven development, experiment control, code generation, sounding rocket

Acronyms/Abbreviations

Application programming interfaces (APIs), Bose-Einstein condensate (BEC), domain-specific language (DSL), experiment execution graph (EEG), Graphical Modelling Framework (GMF), graphical user interface (GUI), interface control documents (ICD), International Space Station (ISS), matter-wave interferometry in microgravity (MAIUS), printed circuit board (PCB), Systems Modelling Language (SysML), telecommand/telemetry (TM/TC), Unified Modeling Language (UML), YAML Ain't Markup Language (YAML).

1. Introduction

A lot of technical challenges arise, when developing software to control a large and complex experiment. One significant problem is the frequently changing constraints and requirements of the software due to the nature of an experimental setup. Experiment sequences and even the hardware will change throughout the

whole development process. Therefore, an important aspect of the development is a high degree of flexibility while keeping the software maintainable. This way new or changed requirements and constraints can be easily integrated, while the software stays stable enough to reliably run the experiments. The classical design process for the software with fixed requirements and designs is often not feasible when the experiment itself has a low technology readiness level at the beginning of the process and the technology extends the boundaries of what is technical possible.

This paper presents the architecture and the software engineering approach of the experiment control software for the MAIUS-1 mission [1]. The MAIUS-1 mission had the goal to conduct cold-gas experiments on a sounding rocket flight and create the first Bose-Einstein condensate (BEC) in space.

An experiment and mission of this size involves many domain experts, each having a specific perspective on the experiment. The biggest challenge we were facing was how to combine all requirements and contributions of the different domains involved into a single flight software product.

An essential aspect of the process was the formalization of the knowledge transfer between the domain experts. It is essential to have a knowledge-base of some kind where everyone can add and change data about their own domain and retrieve information about others.

Usually a knowledge base is written in natural language in form of formal documents or a collaborative wiki. Especially in space projects the first approach is followed. Formal documents with release and approval processes are usually the driver of such large-scale developments.

For the flight software development of MAIUS-1 we set up a model-driven engineering approach that has been widely proposed to replace the currently dominant document-centric engineering approach [2]. We created a machine-readable, formal model of the experiment apparatus and developed the software with the information stored in this model. This way the domain experts themselves can directly contribute to the software without having to care too much about software engineering aspects. From the software domain perspective this approach is advantageous as it enables the automatic generation of parts of the control software using the information embedded in the models.

This paper describes our way to implement these abstractions using a model-driven approach to develop the software. Several domain-specific languages (DSL) then provide different views of the system.

The remainder of the paper is organized as follows: The next section gives a brief overview of related work. Sect. 3 presents the use case of the MAIUS-1 mission in more details. Sect. 4 introduces the concept of the flight software development following the model-driven development approach. Sect. 5 gives an overview of the actual implementation of the flight software followed by Sect. 6 with some results. Finally, Sect. 7 gives some conclusions and an outlook to future work.

2. Related work

Most embedded software in the space domain is written in C. The main reason for this is the flexibility and availability of vendor support. The problem with powerful languages such as C/C++ is the high complexity that comes with the power and in consequence the big effort one must take to keep the software maintainable.

To face the problem of complex programming languages specialized languages have been developed. A sophisticated way to keep the system maintainable and decrease the manual implementation of interfaces is to apply model-driven software architectures [3]. These kinds of architectures rely to a great extent on a model which maps the structure and interfaces of the system. Out of this model, views can be derived to provide all the important information about the system to the different domain experts while hiding unnecessary parts of the system. This makes it possible to describe the software in a way that is focused on the problem and abstracted from implementation details.

To describe these models general-purpose modelling languages or so-called domain-specific languages (DSLs) can be used. The most popular modelling language for the software domain is the Unified Modelling language (UML). For modelling physical systems, an extension to UML, the System Modelling Language (SysML), was created. Both languages provide graphical diagrams to describe certain aspects of the software or the system, respectively. Other common modelling languages can be found in commercial products like Matlab/Simulink.

DSLs on the other hand open the possibility to provide a specialized formal language for a single domain of the project which can be designed to fit their needs. Moreover, it is possible to define multiple DSLs to cover multiple domains with different programming models and merge them into a single model afterwards [4]. The idea behind this concept is the mapping of the problem space into the solution space [5]. With a DSL it is possible to solve tasks in the problem space, which is a view on the system that is fitted to the specific problem. DSLs can be either textual or graphical.

Modelling software for design and documentation purposes has been applied for many years. The large benefit in terms of quality and cost of the software development comes when also automatic code generation from the model is used. This allows the transformation of the model description from the problem space to the solution space. The solution space

finally is usually a general programming language which can then be compiled and executed on the target platform.

Like the modelling approaches, there exist general-purpose code generators and domain-specific code generators. Examples for general code generators are code generators included in UML editors like Enterprise Architect or the autocoders available for Matlab/Simulink. Domain-specific code generators are usually based on coding templates and can be specifically tailored to the specific domain or project [6].

The trend to move from manual written source code to automatically generated code is common in safety-critical embedded systems, especially in the automotive domain [7]. But also in the aerospace sector, autocoding is used more often in recent years. For instance, the Mars Science Laboratory entry descent, and landing flight software consists of approximately 1 million lines of handwritten C code and 2.5 million lines of autocoded software [8].

3. MAIUS-1 mission use case

The use case for which we implemented the model-driven software architecture is the experiment control software of the MAIUS-1 mission. It is tailored to a highly customized hardware setup which uses its own protocol for communication. The following section describes the setup and requirements of the experiment hardware which is important to understand the concepts behind the models.

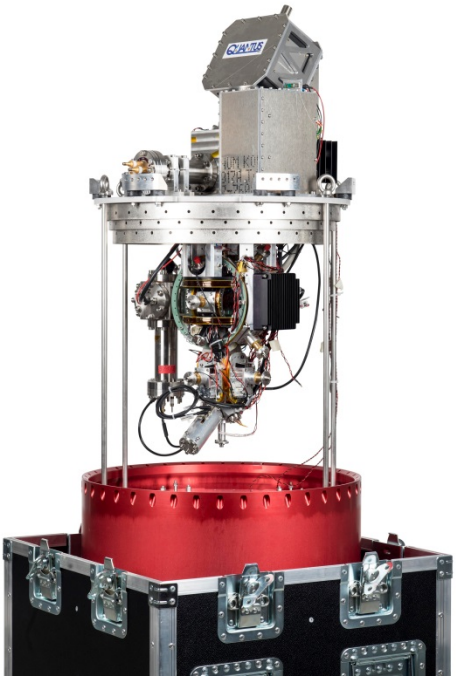

Fig. 1. The experiment apparatus of MAIUS-1 before integration into the rocket (© Leibniz University Hannover)

Fig. 1 shows the MAIUS-1 cold atom experiment before integration into the sounding rocket. The system consists of an ultra-high vacuum system [9] with an atom chip, a diode laser system, and compact electronics, which will be discussed in the following.

3.1 Experiment hardware

The hardware which must be controlled by the on-board software consists of so-called “cards” at the lowest level. Every card is basically a single printed circuit board (PCB) with chips for a specific purpose (e.g. driving laser diodes or control shutters) and is stackable with other cards (see Fig. 2 for an example electronic stack). Those cards are assembled into so-called “stacks”, where each card is connected with the others over a bus. A stack usually has all the cards for a specific subsystem of the experiment (e.g. laser driver or vacuum pump control). The cards in this stack are programmable, meaning that some parameters can be set either directly or programmed in advance to follow a timed order of parameter settings. The programming is done via an Ethernet interface to the on-board computer. These timed instructions which we call “sequences” have to be uploaded to the corresponding stacks and can then be started simultaneously on all stacks. The hardware is then executing the sequences in real-time.

This brief overview of the hardware outlines the components and features the on-board software must address. In terms of model-based software engineering, these components have to be mapped into a model of the hardware, and the on-board software will rely on this model.

3.2 Involved domains

As explained before, the construction of an apparatus of this complexity needs the involvement of experts from different domains. The experiment control system of the MAIUS-1 mission mainly involves three domains: electronics, physics, and software.

The electronics domain is responsible for the hardware design and the definition of the communication protocols to access the hardware. From

Fig. 2. One of the electronic stacks from the MAIUS-1 experiment

the software point of view, they provide the description how to communicate with the cards and stacks they design, and which parameter of each card can be set or read.

The second domain, physics, consists of the people who actually conduct the experiment. They define the scientific goals of the mission as well as the behaviour of the experiment to achieve these goals.

The software domain is responsible for the architecture, development, and integration of the experiment control software.

3.3 Autonomy requirements

It was crucial for the mission that the experiments can be conducted autonomously during the flight without involvement of the ground station. The microgravity phase, where most of the experiments were carried out, was only six minutes long. Constant contact to the ground station could not be guaranteed. Thus, experiments that rely on human interactions were not possible due to the fragility of the radio link, but limited access was possible. Additionally, the limited time of the microgravity phase should be used for as many (different) experiments as possible. To make the most out of this limited time, optimizations and adjustments on the parameters had to be done in-flight as they are directly coupled to the latest measurements of the experiment.

Without autonomous execution of the experiments the mission might fail as parameters cannot adapt to the current state of the apparatus.

As shown, it is necessary that optimizations must be done on the on-board computer instead of sending data down and have the experimenters decide on the experiment flow. However, to keep the control of the experiment flow, in case the autonomous control fails, the control mechanism needs a way to influence the execution from the ground.

3.4 Payload integration

The on-board computer had also the task to communicate with the avionic system of the sounding rocket and to provide a telecommand/telemetry (TM/TC) interface. The integration of the MAIUS-1 payload in the flight system is detailed in [10]. The communication to the service module of the rocket was realised by a special serial interface and an Ethernet link for the TM/TC.

4. Concept

The main design driver of the flight software concept for MAIUS-1 is the creation of a joint model, which describes both the hardware and the experiment flow. The goal was, from the software engineering perspective, to create tools to allow the domain experts to describe the hardware and the experiments, respectively, by themselves.

These descriptions should then automatically be converted to the actual flight software. If all the tools are available, the software developers would be theoretically not needed, when changes are made to the hardware or the experiments are prepared.

However, engineers and scientist from different domains working on the experiment apparatus have their own specific views on the system with a focus on their domain. All these different data need to be combined in the development of the flight software. The challenge was how these different views and requirements could be combined in a consistent way that not a single person or group, for instance the software development team, had to check and manage the complete process manually.

To achieve this, interfaces needed to be established to define the communication between the components of the domains. Traditionally this is done with documents that describe these interfaces and which are then implemented manually. One example for these documents in the area of on-board software for spacecraft are interface control documents (ICD).

In our experience, defining those interfaces in a document tend to result in a lot of overhead work throughout the whole development process. The reason for this is that the documents exist independently of the implemented interfaces and changes in each of them must be synchronized with the others.

Following the model-based approach, the model becomes the single point of truth that collects all information and automatically ensures the consistency between the different components and domains. From this model the actual source code of the flight software is then generated. This way we can use the model itself not only to describe the software that has to be generated, but also use the model as a knowledge base of the system. Once these interfaces are defined, it is possible to provide an interface to the domain experts that will update automatically whenever a part of the model has been changed.

4.1 System layers

To achieve the goal of providing domain-specific views of the system several abstraction layers were introduced. The information from the different layers were combined to a model that covers all important aspects relevant to the flight software and supporting tools. From this model, source code and other artefacts can then be generated.

To enable the domain experts to provide the required information, domain-specific front-ends to this model were developed. The front-end on each layer provides only the necessary technical details for the layer in question.

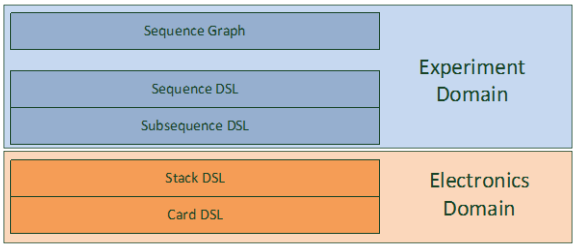

Fig. 3. The conceptual layers of the different DSLs and the targeted domain

In total five different descriptions, i.e. domain-specific languages, have been created for this architecture: four textual and a graphical modelling language. In a bottom-up ordering, starting with the description of the actual hardware and the communication with the on-board computer, each following language will abstract the representation up to a description of the experiment behaviour. These hierarchical layers are shown in Fig. 3 and will be explained in detail in the following subsections.

These descriptions are then combined and converted in a model. From this model, the C++ source code for the flight software as well as descriptions for supporting tools is generated. This includes a channel list for the sequence generation and templates to decode the different telemetry data streams on the ground station. Fig. 4 shows the general workflow of this process.

4.2 Hardware description

The base layer of the model is the description of the hardware. This description represents the communication with the custom experiment hardware in a textual way. The description is split up into two parts with respect to the hardware architecture described in Sect. 3.1: cards and stacks.

The Card DSL is used to describe the parameters which can be read from and written to a hardware card. This includes for example the local address, the data type, and the bit pattern used for a specific parameter. To access these parameters the Card DSL exposes so-called channels to the outside which abstract the access to the parameters. This is needed as some parameters need a sequence of commands to access the correct parameter (e.g. setting a multiplexer before reading out an analogue measurement).

To model the stacks, which aggregate cards, another description has been designed: the Stack DSL. Each instance of a stack holds instances of the cards it contains and addresses them inside the stack bus. Furthermore, the channels, which are provided from the aggregated cards, are exposed and define the feature set of the single stack that is then used by the higher levels of the model.

These two descriptions correspond to the electronic domain point of view to the entire software. Therefore, the DSLs have been designed together with the hardware developers to provide them an intuitive tool to describe their own hardware.

The hardware description provides the basic functionality to control the apparatus. The next layers build upon this foundation and add the behaviour to the model.

4.3 Sequence description

The next two levels in Fig. 3 lead to the experiment domain consisting of the sequence models: subsequences and sequences.

Sequences are an abstract description of the behaviour of a single step of the experiment. One can think of sequences as precisely timed steps of changing the state of the apparatus. To achieve this, the following information is needed: which channel should be modified, how the channel is modified, and when to change it. For the parameters, the sequence description will import all channels exported from the stack description that have been set up for the apparatus, so that one can choose from all the available channels described in the hardware description. Multiple channel

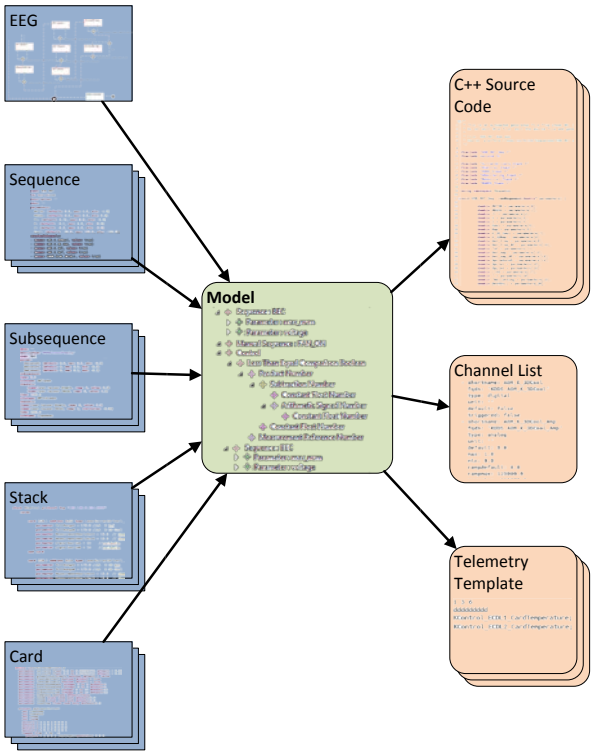

Fig. 4. The model-driven workflow from the different abstraction layers to the generated source code of the flight software and additional artefacts

settings can then be grouped into time slots, which define the exact time when to set the given set of parameter changes.

Additionally, to prevent repetitive code, the description of the sequences is subdivided into subsequence code as many parts of sequence will be reused. Subsequences and sequences support parameters in time and values. This makes both DSLs a very versatile tool to implement the experiments.

These two layers represent the behaviour of the experiment hardware and maps to the real-time programming technique of the cards and stacks.

The Sequence and Subsequence DSL are one part of the physicist domain view to the system giving the experimenters a textual description of the single steps of the experiment flow.

We designed both sequence DSLs with a tailored YAML (YAML Ain't Markup Language) [11] syntax to find a good trade-off between human and machine readability. To create the sequences and subsequences, a special graphical user interface (GUI) was developed to support the creation of all required sequences (see Fig. 5). This GUI exported the sequences and subsequences in the defined YAML format.

4.4 Experiment execution graph

The uppermost layer of the model (see Fig. 3) is a graphical representation of the experiment flow called the Experiment Execution Graph (EEG). Basically, it merges all information provided in all the lower layers into a big picture. It provides methods to order the sequences as well as setting up decisions and branch the execution depending on the state of the experiment (e.g. measured values or internal states of the software).

As seen in Fig. 6, the flow of execution is designed as a binary decision graph. Like UML activity diagrams, sequences are represented by boxes and decision points by diamond shapes.

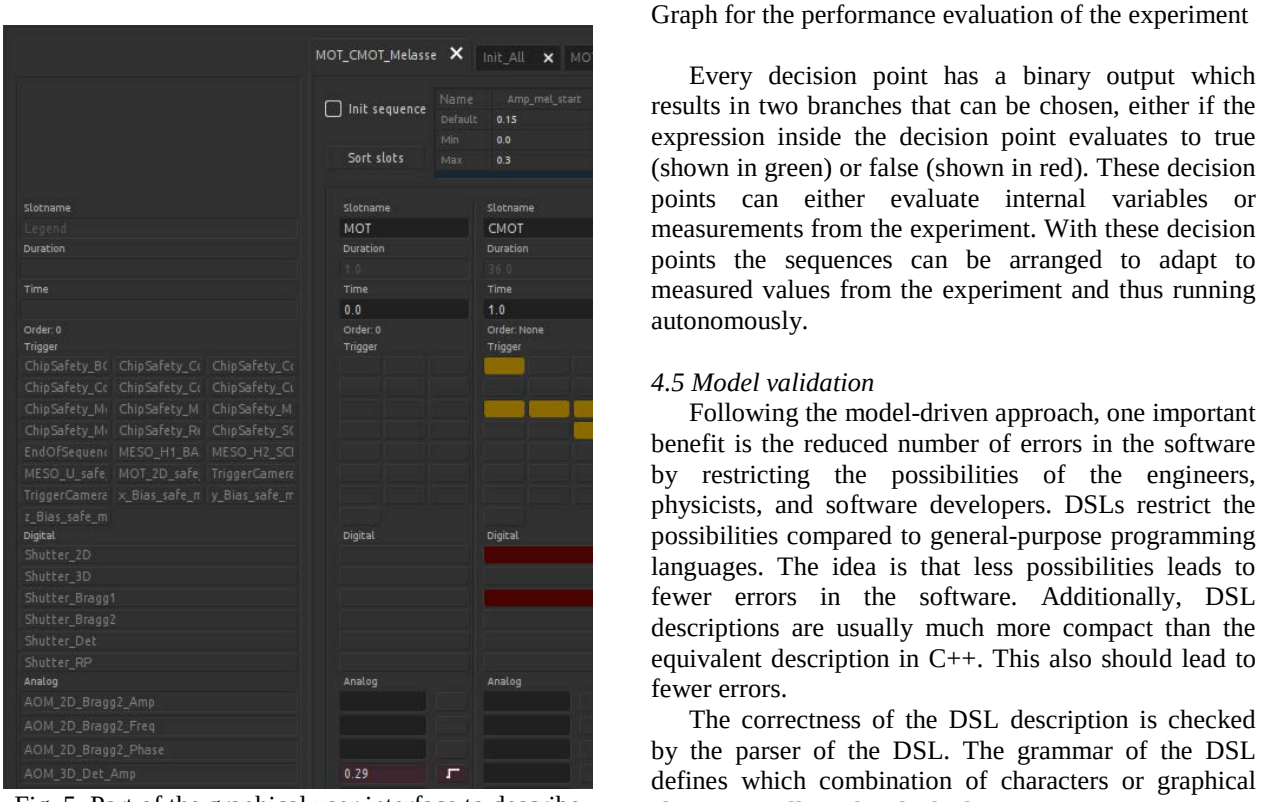

Fig. 5. Part of the graphical user interface to describe sequences and subsequences.

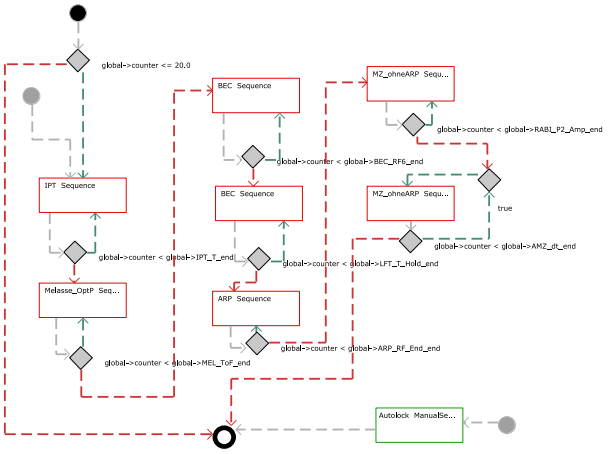

Fig. 6. An example of an Experiment Execution Graph for the performance evaluation of the experiment

Every decision point has a binary output which results in two branches that can be chosen, either if the expression inside the decision point evaluates to true (shown in green) or false (shown in red). These decision points can either evaluate internal variables or measurements from the experiment. With these decision points the sequences can be arranged to adapt to measured values from the experiment and thus running autonomously.

4.5 Model validation

Following the model-driven approach, one important benefit is the reduced number of errors in the software by restricting the possibilities of the engineers, physicists, and software developers. DSLs restrict the possibilities compared to general-purpose programming languages. The idea is that less possibilities leads to fewer errors in the software. Additionally, DSL descriptions are usually much more compact than the equivalent description in C++. This also should lead to fewer errors.

The correctness of the DSL description is checked by the parser of the DSL. The grammar of the DSL defines which combination of characters or graphical elements is allowed and which not.

Besides the check of the syntactical correctness of the models we also designed validators that checked other semantic aspects. For example, the feasibility of the duration of time slots for sequences was checked or that the minimum value of a parameter is actually lower than the maximum value.

These validators increase the reliability of the system, since it ensures the correctness already in the description phase.

5. Implementation

The architectural design of the software architecture is highly focused on modularity to keep the software as flexible as possible. Fig. 7 shows the top-level components of the software architecture. Each blue box represents a single module in the software where the orange boxes stand for hardware components that must be accessed and controlled from the software. The software modules are designed as stand-alone components, which run in their own tasks. Every module has defined input and output slots to communicate with other modules. This way the side-effects were kept to a minimum and only appear at the hardware communication.

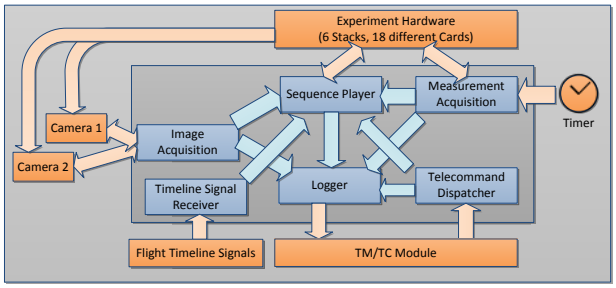

Fig. 7. The components and peripherals of the MAIUS-1 flight software

For the communication between these modules, so-called channels are provided (shown as arrows in the figure). Each channel is unidirectional and handles a single datatype that has to fit the input and output slots of the connected modules.

As the basis for this architecture DLR's in-house development "Tasking Framework" [12] was used. It provides the base classes for the tasks, the communication channels, and a scheduler to ensure the timely execution of each task.

5.1 Software components

The main software components, as seen in Fig. 7, are responsible for controlling the complete experiment.

The *sequence player* is organized as a play-list-based music player where the music titles are in this case the experiment sequences. In the lab, the operator can load lists of sequences that are executed by software. Sequences can be added or removed, and the order can be changed by telecommands. When the experiment is control by the EEG, the graph controls the sequence player. If, for instance, at a decision point a different sequence needs to be executed, the graph loads the sequence to the sequence play list.

The *logger* is responsible of storing all measurements locally on the on-board computer and compressing data for the telemetry link. The data includes experimental values and camera images as well as housekeeping data of the experiment. The architecture of the logger is quite generic. One can define filters and data writers. It can be configured, which data should be filtered and written to which target. This allows a fine-grained configuration of the data handling.

The *measurement acquisition* is the component that regularly accesses the experiment hardware for sensor readings and forwards it to the logger and some to the sequence player for decisions in the EEG. The *image acquisition* handles the capture and sending of camera images from the experiment.

The *telecommand dispatcher* receives the telecommands from the ground operator and distributes them through the system, for example, to the sequence player.

5.2 Integration of the models

The integration of the different models is organized in layers according to the different model layers described in Sect. 4.

Fig. 8 gives an overview of the different layers. The base layer for the whole integration is the execution platform with the core components of the system, which provides a common interface to implement the artefacts of the upper layers and provide methods to manage these artefacts. We used the Linux Operating System since the real-time requirements are not strict. The custom-build hardware takes care of the real-time execution of the sequences as described in Sect. 3.1.

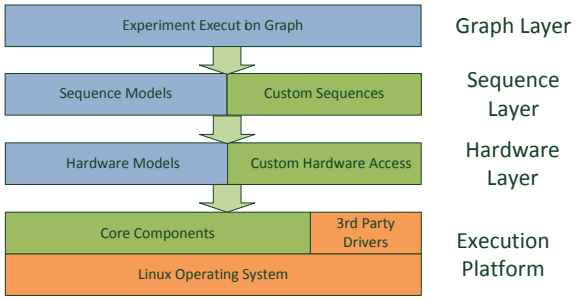

Fig. 8. The different layers of the model integration

Further components on the base layer, like Tasking Framework, Sequence Player, etc. are written in C++; this implies that the whole integration was done in the

C++ language. That means software artefacts can be written either in C++ directly or generated from a model using a C++ code generator for that model.

The next two layers are corresponding to the hardware and sequence models. These artefacts can be either generated from the models described by the according DSLs or implemented in C++ following the defined application programming interfaces (APIs). On the hardware layer, these custom implemented artefacts can be used to access hardware that is not part of the cards and stack hardware (e.g. cameras).

The same applies to the sequence layer, whose artefacts can be generated from the models of Subsequence and Sequence DSL or manually using C++. A manual sequence is, for example, a module to evaluate camera images to automatically detect features of the cold atoms.

The topmost sequence graph layer then is purely generated from the graphical sequence graph DSL, the EEG. The integration is loosely coupled at the interfaces between the different layers and can be used standalone for specific use cases where only parts of the experiment are used in the lab. This makes it possible, for example, to explicitly set parameters using the interface of the hardware model or executing sequences without creating a whole graph.

5.3 Modelling framework

The DSLs, validators, and code generators are implemented on the Eclipse Rich Client Platform. Specifically, the Xtext framework [13] provides the basis to create textual DSLs by defining the grammar. Xtext provides then the infrastructure: editors with syntax highlighting and a framework to implement validators.

The code generators are implemented with the Xtend language within the Xtext framework. The graphical editor for the EEG was created with the Eclipse Graphical Modelling Framework (GMF).

This whole toolset was integrated in a single software packet that was provided to the engineering and scientist team as a graphical user interface for creating the different models. Another instance of the code generators was created to provide automatic code generation in the build process of the software.

6. Results

For the use case of the MAIUS-1 mission, the development technique presented in this paper has been successfully deployed. MAIUS-1 was launched in January 2017 from Esrange in Kiruna, Sweden. Fig. 9 shows the MAIUS-1 team before the final integration of the experiment with the rocket. MAIUS-1 created the first BEC in space and successfully conducted experiments with it [14].

In the final flight software, 84% of the C++ code was generated from the models. The generated code covers all boilerplate code that otherwise had to be implemented by hand. As an added effect, the generation of the boilerplate code increased the overall code quality as these code segments are repetitive and writing them by hand is error-prone and hard to debug.

Fig. 9. MAIUS-1 team with the payload and the service module before integration with the rocket (© T. Schleuß)

However, we do not have statistics to proof this claim, since we did not implement the software twice with different methods for comparison. Nevertheless, our experience with manual software development in several space missions gives us confidence to support this claim.

Beside the software engineering aspects, the use of a combined model to describe the experiment hardware and sequences had a positive effect on the collaboration of the different domain experts. Every domain involved in the development of the control software had their own specialized view to the system on which they can work. This approach was accepted and appreciated by the software developers, engineers, and scientists.

The process of the definition and design of the DSLs was a major step in the project to build a collective understanding between the different domain experts. Designing a formal language turned out to be more efficient than trying to write long lists of requirements. The grammars of the DSLs can be seen as formalized requirements. A missing feature in the one of the DSLs meant a not communicated requirement or a misunderstanding between the domain experts and the software development team.

The whole software was very flexible as well. With only few modifications, it was used to control a

secondary payload of the MAIUS-1 mission. This payload tested some new laser technologies and was controlled by the same software on a computer independent from the MAIUS experiment.

However, the initial requirement to develop software for a short flight turned out to be cumbersome during lab tests. We decided at the beginning to generate the whole flight software and do not work with interpreted code or similar approaches to make the software more reliable. The drawback of this approach is that each change of a sequence needs a recompiling and restarting step.

7. Conclusions and outlook

This paper presented the model-driven software engineering approach for the complex experiment control of the MAIUS-1 mission, which created the first BEC in space on a sounding rocket. The model-driven approach was successfully implemented. Besides the higher software quality as a result, the process of defining domain-specific formal languages for the different domains involved turned out to be an efficient tool to gather and validate requirements instead of writing long requirement documents.

Two MAIUS flights are planned for the next years and an experiment on the International Space Station (ISS) is currently prepared. This experiment, called BECCAL, and the sounding rocket missions will be controlled by improved versions of the presented software. The biggest change will be an interpreted sequence player. The subsequences, sequences, and EEGs will no longer be transformed to C++ code but to an intermediate representation, which will be executed by a custom interpreter. This allows the modification of sequences and EEGs without recompiling and restarting the flight software.

Acknowledgements

This work is supported by the DLR Space Administration with funds provided by the Federal Ministry of Economics and Technology (BMWi) under grant number DLR 50WM1131-1137, DLR 50WM1552-1557, DLR 50WP1431-1435, DLR 50WM0940, and DLR 50WM1240. We thank the MAIUS-1 team for their contributions and support. In particular, we want to mention H. Ahlers, D. Becker, A. N. Dinkelaker, M. D. Lachmann, S. T. Seidel, and E. M. Rasel. We are grateful to our former colleague M. Deshmukh for her contributions to the DSL development.

References

[1] S. T. Seidel, M. D. Lachmann, D. Becker, J. Grosse, M. A. Popp, J. B. Wang, T. Wendrich, E. M. Rasel, Quantus Collaboration, Atom Interferometry on Sounding Rockets, pp. 309–312, Proceedings of the 22nd ESA Symposium on European Rocket and Balloon Programmes and Related Research, Tromso, Norway, 2015, 7 – 12 June.

[2] A. L. Ramos, J. V. Ferreira, J. Barceló, Model-Based Systems Engineering: An Emerging Approach for Modern Systems, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 42.1 (2012) 101–111.

[3] B. Selic, The pragmatics of model-driven development. IEEE Software, 20.5 (2003) 19–25.

[4] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, K. Olukotun, Composition and Reuse with Compiled Domain-Specific Languages, pp. 52–78, ECOOP 2013 -- Object-Oriented Programming: 27th European Conference, Montpellier, France, 2013, 1 – 5 July.

[5] K. Czarnecki, Overview of generative software development, in: J.-P. Banâtre, P. Fradet, J.-L. Giavitto, O. Michel (Eds.), Unconventional Programming Paradigms: International Workshop UPP 2004, Le Mont Saint Michel, France, September 15 – 17, 2004, Revised Selected and Invited Papers, Springer Berlin Heidelberg, 2005, pp. 326–341.

[6] T. Franz, D. Lüdtke, O. Maibaum, A. Gerndt, Model-based software engineering for an optical navigation system for spacecraft, CEAS Space J, 10.147 (2018) 147–156.

[7] C. Ebert and C. Jones, Embedded Software: Facts, Figures, and Future, Computer 42.4 (2009) 42–52.

[8] K. P. Gostelow, The Mars Science Laboratory entry, descent, and landing flight software, 23rd AAS/AIAA Spaceflight Mechanics Meeting, Kauai, Hawaii, 2013, 10 – 14 February.

[9] J. Grosse, S. T. Seidel, M. Scharringhausen, C. Braxmaier, E. M. Rasel, Design and qualification of an UHV system for operation on sounding rockets, J. Vacuum Science & Technology A 34 (2016) 031606.

[10] A. Stamminger, J. Ettl, J. Grosse, M. Hörschgen-Eggers, W. Jung, A. Kallenbach, G. Raith, W. Saedtler, S. Seidel, J. Turner, M. Wittkamp, MAIUS-1 - Vehicle, subsystems design and mission operations, pp. 183–190, Proceedings of the 22nd ESA Symposium on European Rocket and Balloon Programmes and Related Research, Tromso, Norway, 2015, 7 – 12 June.

[11] O. Ben-Kiki, C. Evans, B. Ingerson, YAML Ain't Markup Language (YAML™) version 1.2, http://www.yaml.org/spec/1.2/spec.html, (accessed 13.09.18).

[12] O. Maibaum, D. Lüdtke, A. Gerndt, Tasking Framework: Parallelization of Computations in Onboard Control Systems, ITG/GI

Fachgruppentreffen Betriebssysteme, Berlin, Germany, 2013, 7 – 8 November.

[13] M. Eysholdt, H. Behrens, Xtext: implement your language faster than the quick and dirty way, pp. 307–309, Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, Reno/Tahoe, Nevada, USA, 2013, 17 – 21 October.

[14] 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, D. Lüdtke, T. Franz, N. Gaaloul, W. Herr, 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, E. M. Rasel, Space-borne Bose-Einstein condensation for precision interferometry, eprint arXiv:1806.06679, 2018.