MODEL-BASED SOFTWARE ENGINEERING FOR AN OPTICAL NAVIGATION SYSTEM FOR SPACECRAFT[1]

T. Franz, D. Lüdtke, O. Maibaum, A. Gerndt,

German Aerospace Center (DLR), Simulation and Software Technology,
Lilienthalplatz 7, 38108 Braunschweig, Germany

Abstract

The project ATON (Autonomous Terrain-based Optical Navigation) at the German Aerospace Center (DLR) is developing an optical navigation system for future landing missions on celestial bodies such as the Moon or asteroids. Image data obtained by optical sensors can be used for autonomous determination of the spacecraft's position and attitude. Camera-in-the-loop experiments in the TRON (Testbed for Robotic Optical Navigation) laboratory and flight campaigns with unmanned aerial vehicle (UAV) are performed to gather flight data for further development and to test the system in a closed-loop scenario. The software modules are executed in the C++ Tasking Framework that provides the means to concurrently run the modules in separated tasks, send messages between tasks, and schedule task execution based on events. Since the project is developed in collaboration with several institutes in different domains at DLR, clearly defined and well-documented interfaces are necessary. Preventing misconceptions caused by differences between various development philosophies and standards turned out to be challenging. After the first development cycles with manual Interface Control Documents (ICD) and manual implementation of the complex interactions between modules, we switched to a model-based approach. The ATON model covers a graphical description of the modules, their parameters and communication patterns. Type and consistency checks on this formal level help to reduce errors in the system. The model enables the generation of interfaces and unified data types as well as their documentation. Furthermore, the C++ code for the exchange of data between the modules and the scheduling of the software tasks is created automatically. With this approach, changing the data flow in the system or adding additional components (e.g. a second camera) have become trivial.

KEYWORDS

Model-driven software development, Model-based systems engineering, Code generation, Optical navigation,

1. INTRODUCTION

Future mission designs for the robotic exploration of celestial bodies require the landing of scientific instruments at specific locations with a high accuracy. The project Autonomous Terrain-based Optical Navigation (ATON) at the German Aerospace Center (DLR) develops methods to use optical sensors to compute a navigation solution in real-time. Such dynamic methods provide a way to handle unexpected situations where a static command list would fail. This increases the achievable accuracy to reach the destined landing site. Optical systems for navigation are a promising technology since their measurements are independent from a ground station.

The project has been running since January 2010. It has passed several stages of simulation and flight tests. Sensor simulations and camera-in-the-loop experiments in the TRON (Testbed for Robotic Optical Navigation) laboratory were used to test the system with close to realistic scenarios. A camera is installed on a robot and moved over different surface models, representing the different phases of a lunar landing. In addition, flight tests with unmanned aerial vehicles are performed to demonstrate the robustness and accuracy of the system in closed-loop scenarios.

The optical sensors consist of two cameras, a star tracker and a laser altimeter. Besides that, the system uses an inertial measurement unit (IMU). The collected data is analyzed in several intermediate steps and finally fused in a Kalman filter to estimate position and attitude of the spacecraft [1]. The camera data is used to compute a relative movement by using feature tracking [2] as well as position estimations by matching shadows [3] and

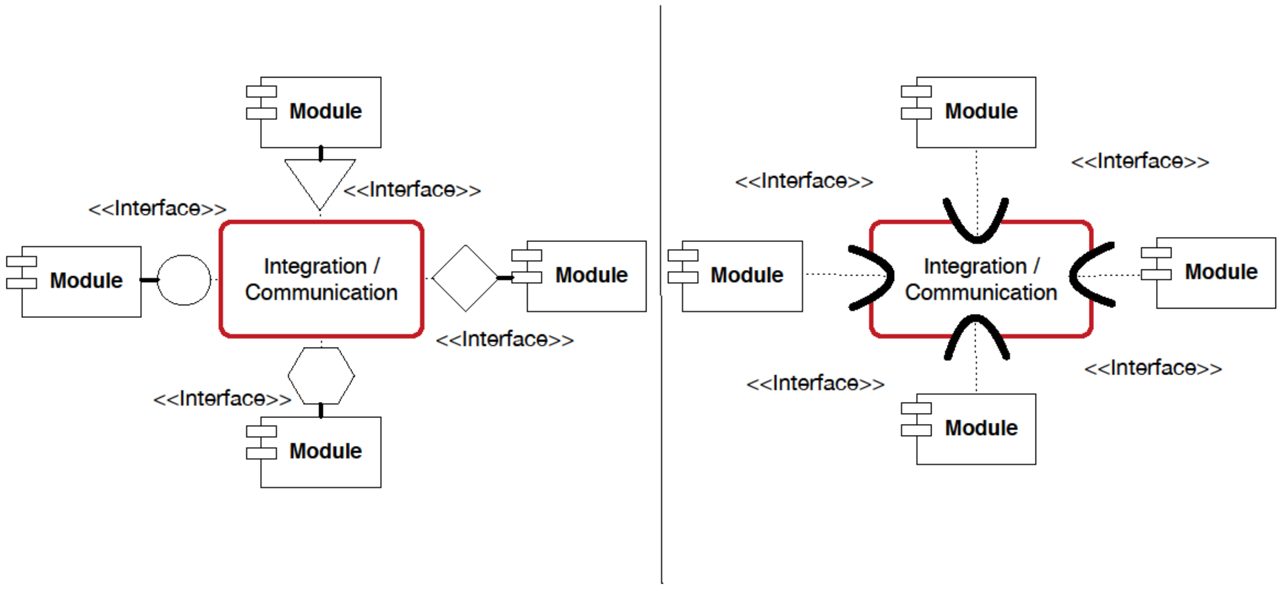

FIGURE 1 Interface communication before (left) and after (right) the introduction of a system model to the software development. The central definition in a model leads to a shared datatypes over all interfaces. Code generation unifies structure and coding style.

craters [4].

Due to the high complexity and specialization of the used algorithms, the software modules are developed at different DLR research institutes from different domains (space systems, robotics, optical systems, aeronautic flight systems and simulation and software technology). The integration of all those modules into one software turned out to be very challenging.

Even with clearly defined software interfaces in an Interface Control Document (ICD), some misunderstandings occurred. Since the corresponding module developers provided the interfaces, data types had different formats and needed to be converted during integration. Moreover, different institutes and development teams used varying coding styles, guidelines and even different programming languages because not all algorithms were developed solely for this project.

Additionally, the integration of the software modules into the execution platform was very time-consuming. To be executed in the system's scheduler, each module needs to be encapsulated into a special container that handles communication with other modules.

Besides the actual integration of a module, changes in an interface caused much work as well. The integration team needed to document the modification in the ICD and had to update all related parts of the software.

To overcome the before mentioned integration problems, we introduced a model-based development approach, which is presented in this paper. The general idea of Model-based Systems Engineering (MBSE) is to collect all required information for information exchange between engineers in a central model rather than using documents. Instead of trying to combine interfaces implemented independently, the coherence of the components and the software's internal interfaces should be defined from a system point-of-view. Defining this kind of information in a formal model enables analysis and reduces misunderstandings between all involved parties [5].

Models can be used to support design, analysis and validation activities even before the software implementation. For instance, the compatibility of inputs and outputs of communicating software modules can only be checked manually with the interface control document. If a formal model is used to define inputs and outputs, a software validator can automatically check for compatibility. Since modeling environments are usually based on standardized concepts like the Meta-Object Facility (MOF), models can be analyzed, validated and transformed with existing tools [6].

Figure 1 shows the difference before and after the introduction of MBSE to the ATON software framework. During the manual integration, module developers defined the module interfaces and the integration and communication code needed to contain the conversion of data types. With MBSE, the generator creates the module's interface code using shared data types defined in a central model.

This paper describes the application of model-based techniques for the development of embedded real-time systems. We apply the methodologies of model-based systems engineering (MBSE) to the development of software for an optical navigation system. The goal is to specify the system design in a model and to use it to analyze and unify the software interfaces. Additionally, we use the concepts of model-driven software development (MDSD) to reduce the overhead for the integration of new system components.

We organized the remainder of the paper as follows. Section 2 gives an overview of related work. In Section 3, we introduce the MBSE concept for ATON, followed by a brief overview of the implementation. Section 5 evaluates our approach. Finally, Section 6 gives some conclusions and an outlook to future work.

2. RELATED WORK

Space systems have some specific requirements for the onboard software regarding reliability due to the harsh space environment. In addition, it is usually not possible to reach such systems for maintenance after launch. This led to the development of several modeling tools, environments and languages targeting embedded systems and space systems in particular.

The European Space Agency (ESA) develops a complete toolchain for model-driven software development called TASTE (The ASSERT Set of Tools for Engineering) [7]. It is a set of open source tools for developing embedded real-time systems for space missions. It focuses on error-prone processes such as the integration of subcomponents. TASTE relies mainly on two complementary modeling languages, one specifying the data structures, the other describing the software architecture. The philosophy of the tool chain is to let the user concentrate on the functional code and let TASTE handle the integration. The tool is also able to combine software components implemented in different programming languages. TASTE generates software that can be directly executed without any further integration. Two concepts of TASTE are of particular interest:

- Describing the system with two complementary viewpoints, one graphical description for the software architecture and one for the data structures.
- Generating not just software but also documentation, tests and the build system that makes instant execution possible and decreases the overhead for changes.

Another MBSE approach from the space domain is the CubeSat Reference Model developed by the INCOSE Space Systems Working Group (SSWG) [8]. The CubeSat project was established to reduce costs for small satellite missions by using mainly commercial off-the-shelf components. The lower budget requirements enable more organizations to develop CubeSats. The reference model has the goal to serve as a guide and supporting tool during development. The CubeSat Reference Model is focused on the demonstration of the model-based methodology for the space system development. The SSWG proposes that besides the engineering methodology, MBSE consists of a modeling language, modeling tools and interfaces to other models. The reference model is implemented using the Systems Modeling Language (SysML). While in this case, SysML is mainly used to capture costs, requirements and life cycle aspects, the language can also be used for formal model checking [9]. Therefore, the semantics are formalized with Petri Nets. In combination with UML profiles, like the Modeling and Analysis of Real-Time and Embedded systems (MARTE), it is possible to do time and schedulability analysis [10].

Because SysML was developed to cover a wide range of scopes related to systems engineering, the language contains a high number of elements and concepts. Consequently, the effort to learn the necessary parts of the language is high. Additionally, the scope as general-purpose language for systems engineering means a low specialization of its elements. This way, it is possible to stay compatible to different domains. The diagram representations via boxes and lines are very simple and the elements only have generic parameters.

The Jet Propulsion Laboratory investigated methodologies to reduce the learning overhead of SysML and simultaneously add domain specific contents to the model [11]. They suggest building domain-specific languages based on SysML. A domain-specific language (DSL) is a language especially developed for a specific application. To create a language based on SysML, it needs to be customized. SysML can be modified by creating profiles. SysML itself is a profile of the Unified Modeling Language (UML) [12]. Unlike SysML, UML focuses mainly on software. However, Figure 2 shows that both languages are overlapping [13]. While software classes are part of UML, system Blocks are part of SysML. UML's extension mechanism is implemented using ontological concepts, and thereby, the meta-model of UML does not need to be changed [14]. This allows customizations to be applied to a language without the need to adopt its tools like editors.

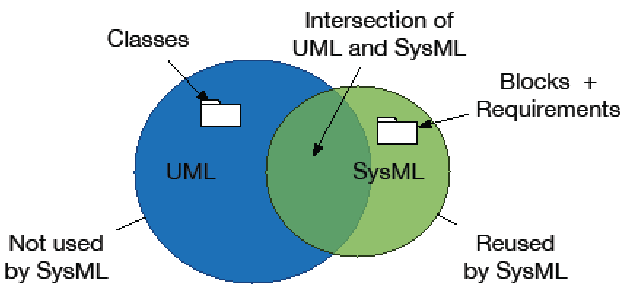

FIGURE 2 Illustration of the relation between UML and SysML. The concept of classes is contained in UML, Blocks are part of SysML. Figure inspired by Iqbal et al. [13].

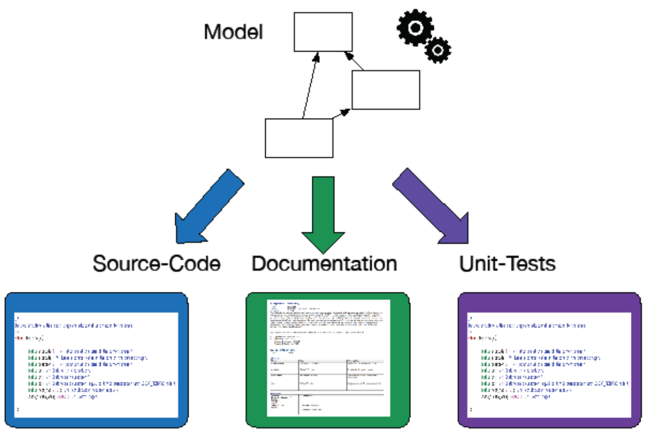

FIGURE 3 Workflow of model-driven development: a model is used to generate project artefacts from it.

In the context of model-based engineering, a model is usually based on the Meta-Object Facility (MOF) [6]. Modeling languages are defined in meta-models, which describe the elements of the domain model. Since the meta-model is a model itself, a root language is necessary. The MOF provides such a language by defining the basic concepts of modeling languages. To create instances of models based on the MOF, the XMI (XML Metadata Interchange) has been introduced as a standard [15].

Model-based systems engineering (MBSE) is the process of collecting all system related information in a central model. Besides information exchange, the model can be used to run analysis and verification. Another concept is the generation of source code from the model. Generators can even create mission-critical code. For instance, the onboard software of NASA's Mars Science Laboratory consists of about 75 percent of generated source code [16]. As Figure 3 shows, model-driven development refers to the process of formally describing a system in a model and then generating project artefacts from it [17].

Model-driven software development (MDSD) is part of the model-driven development and uses the model to generate source code, documentation and unit-tests. The main motivation of using model-driven development is to increase productivity [14]. Short-term productivity is improved by generating new features from the model. Long-term productivity rises because changes of requirements can be easily handled by changing the model. To avoid models with redundant content, systems are described independently from target platform and desired programming language [17].

A framework that is based on model-driven software development should define a set of requirements [14]. It is necessary to define which modeling concepts are used and how the applied model elements are presented as well as their relation to real-world objects. Furthermore, concepts for model extension, interchange and mapping to other project artefacts should be defined.

3. CONCEPT

To overcome integration problems and to reduce the overhead for adding and removing new software elements, we decided to apply the ideas of model-driven development to the structural parts of the software. Because most algorithms were already available as software modules when MDSD was introduced, we did not describe their behavior within the model.

Following the ideas of Atkinson and Kühne [12], an environment for model-driven development should define some basic concepts. This section starts with an analysis of the software, on which MDSD is applied to. This is followed by concepts of the system description.

To be able to navigate the spacecraft with support of optical sensors, the navigation solution has to be computed in real-time. Therefore, all used software modules have strong timing constraints. To reduce communication overhead, the software uses an event-driven approach [18]. If a software module has finished its calculation, all succeeding modules are activated that work on that output data.

To compute a navigation solution from the sensor inputs, the software architecture is data flow oriented. The data is processed and analyzed in several computation steps.

3.1. Functional Requirements

We identified a set of features that a MBSE toolchain needs to provide in the project ATON. Structural models as well as the source code and documentation generators need to be able to specify:

1) System components (processing units + sensors)
2) Data structures passed between the components
3) Inputs and outputs of the software modules
4) Module parameters
5) Scheduling properties (e.g. priority of the tasks)
6) Notification configuration (events and timing)
7) Execution node and thread pool of components

Most of the defined software components have inputs and outputs as well as parameters. To avoid systems with a widely spread or hardcoded configuration, the toolchain should provide a configuration management system.

The software is a real-time system. A customized scheduler executes all components. The model needs to contain information about the priorities of each module.

To take full advantage of the event-driven architecture, the notifications and events have to be configured individually for each software module.

The framework used for scheduling and concurrent execution, supports distributed systems. Software components can be pinned to specific thread pools and it is planned to also use different hardware nodes in the future.

3.2. Non-functional Requirements

Beside these functional demands, there are also some non-functional requirements:

a) C++ code for embedded systems
b) Standard model format XMI

Onboard software for space systems is developed in different programming languages. The most common ones are Ada and C as well as C++. The code generator should be flexible in regards to the target language. The project ATON is mainly developed in C++ with some restrictions for embedded systems. These restrictions refer mainly to the usage of dynamic memory.

In addition to requirements regarding the target language, there are also some model-related premises. To support interchangeability with other tools and environments, the model representation should be based on the XMI standard. One of the most common modeling environments based on XMI is the Eclipse Modeling Framework (EMF) [19]. Since EMF is an open source project, it is the foundation for a large number of tools for both textual and graphical languages.

3.3. Modeling Approach

The initial approach to combine the different algorithms was to use MATLAB Simulink. We modeled the navigation system with a block diagram and integrated software modules via S-functions. This approach was used to develop and adapt the different software modules in a sensor simulation environment. The behavior of the modules was not modeled, as already mentioned, since the algorithms were partially developed in other contexts. While the MATLAB tool suite supports the integration of software modules by adding them to the model, it is hard to customize their generated integration code. To implement the event-driven scheduling mentioned above, we created a new code generator, based on a custom model. This approach allowed us to develop and use domain-specific frameworks and to react flexibly on changes of requirements. Furthermore, our custom code generator has no dependency to MATLAB and thus improves maintainability.

Generating C++ classes from the system model addresses the goal of achieving consistent interfaces. In addition, it can reduce the overhead for adding new modules into the system, since the boilerplate code is automatically generated. Moreover, the source code to establish communication channels and execution containers reduce the development overhead significantly. Especially if new software modules are added or removed, no manual coding is necessary to adapt the communication in the system. To generate this kind of code, the model needs to represent the communication and its parameters. Usually, diagrams are used to represent this information for complex systems. Simple concepts like blocks and lines are easy to understand and directly depict the data flow.

To model all mentioned aspects of the system, one graphical representation is not sufficient. While the data structures and parameters are software implementation details, the components belong to the system design. Recalling the concept of using complementary languages to describe a system (see Section 2); a combination of languages might be a good choice. While SysML is suitable to describe the general system structure, datatypes and parameters can efficiently be defined using SysML's base language UML. Considering that, the Object Management Group (OMG), which was partly responsible for the definition of UML and SysML, underlines that UML and SysML are complementary enough to be used together. In addition, their shared meta-model, infrastructure and tool support are good arguments to use a combination of both languages. In fact, diagrams of both languages can be added to the same model and thus, no model merging is necessary.

Using a combination of SysML and UML covers all views to describe the system but increases the learning effort. The modeler needs to know two languages. However, special system parameters cannot be defined in the model with just standard SysML and UML. Following the approach of a domain-specific language based on SysML, the solution is to create a UML profile covering the custom parameters.

Thus, to provide an efficient modeling tool for our use case, we developed a domain-specific modeling language based on SysML and UML. By using UML profiles for the domain-specific parts, the resulting language is still valid UML/SysML. The next section introduces this language in detail.

4. IMPLEMENTATION

In this section, the two main steps of the model-driven development process are presented. The first step is comprised of the modeling part, where the model is

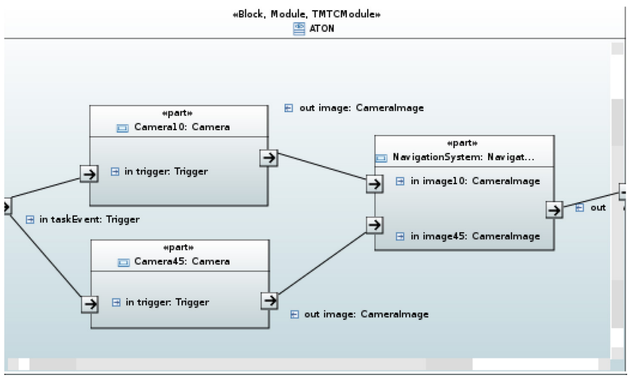

FIGURE 4 A simple example of an internal block diagram: an external trigger synchronizes two cameras. The navigation system computes a navigation solution and provides it to an external interface.

created by describing the system, its communication topology and its parameters. The second step includes the generation part of the process.

4.1. Modeling

The model is a formal abstraction of the system under development. It is the basis for building the system architecture and the generation of source code as well as documentation. XMI orders the model contents hierarchical and can contain multiple diagrams. Diagrams provide different views of the system. UML and SysML editors do the actual graphical representation. We used Papyrus, an Eclipse-based editor [20]. It creates the models by using a native UML implementation, which is based on the Eclipse Modeling Framework. EMF uses XMI, thus requirement (b) is fulfilled.

The main diagram provides an overview on the system and describes the data flow. It is implemented using an internal block diagram offered by SysML. The root element of such a diagram is a block representing the system. Subcomponents are added to the model as a block while their diagram representations are parts. Separating between a block and a part brings an important instance-of relation. One software module can be added twice to a diagram, consuming different input values.

Figure 4 shows a simple system with two cameras, sending their images to a navigation system. The navigation system computes a navigation solution and provides it as an external interface to the system. The small squares appended to the parts are ports. Ports represent a communication endpoint between two elements. As mentioned before, all data passed between elements, executed in the event-driven execution environment, has to be delivered via special containers. These containers are represented as ports. Ports can have a type and other communication related parameters, which are necessary to generate their source code.

While the internal block diagram provides an overview over the software and its communication, the actual data structures are specified in another diagram. Data types are simple classes with attributes, so UML class diagrams are an appropriate way to define them. Besides the data structures, the parameters of the software modules are modeled using class diagrams.

With the definition of ingoing and outgoing data, their structures and the component parameters, all information for the internal interfaces are modeled and no additional explicit definition of interfaces is necessary.

The assignment of software modules to thread pools is modeled in another diagram: the UML deployment diagram. Figure 5 shows an example. The execution environments are modeled as devices, the modules as artefacts. The deployment relation does the assignment of a module to an execution environment.

Besides the general structure specified by UML and SysML diagrams, the special notification and execution parameters have to be added to the model. To create a profile for language extensions, a UML profile diagram has to be created. The element to be customized has to be imported as a meta-class and can then be extended by a stereotype. Stereotypes are classifiers which can contain tagged values, constraints and custom icons. Parameters like the component priority and notification configuration are added as tagged values. Figure 6 shows the extension of a port. The Object Constraint Language (OCL) can be used to define constraints and thus enables model validation.

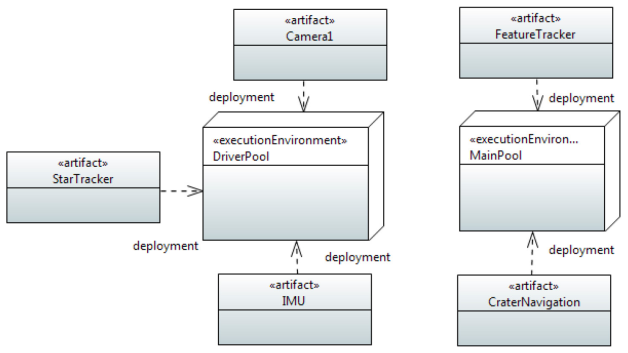

FIGURE 5 Example of a deployment diagram: The IMU, laser altimeter and camera driver are here executed in a thread-pool, the feature tracker and the crater navigation in another.

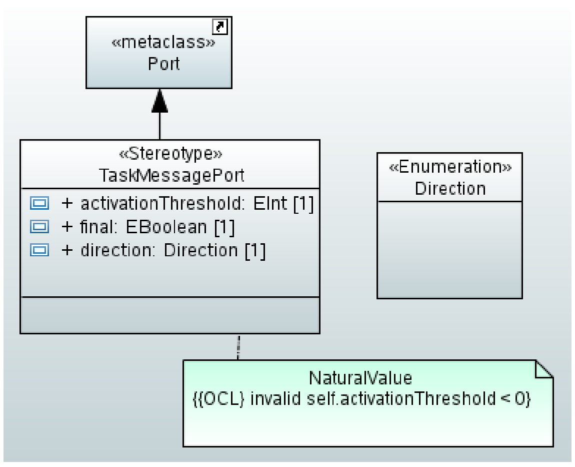
«metaclass»
Port
«Stereotype»
TaskMessagePort
+ activationThreshold: EInt [1]
+ final: EBoolean [1]
+ direction: Direction [1]
«Enumeration»
Direction
NaturalValue
{{OCL} invalid self.activationThreshold < 0}

FIGURE 6 Definition of a customized port: The TaskMessagePort extends a port by adding notification parameters. The constraint checks that all instances have an activationThreshold greater than zero.

4.2. Generation

After the creation of the system model, it can be used to derive project artifacts from it. The generation is done with Xtend, a language compiled to Java and providing a powerful template feature [21]. The syntax is intuitive and it is possible to debug the templates comprehensively. The creation of custom templates enables the generation of source code for any language and thus satisfies requirement (a). Besides source code, it is possible to generate other text files, for example documentation.

The source code generator is implemented using a combination of the decorator pattern [22] and the generation gap pattern [23]. The motivation to unify interfaces and to reduce the overhead for new modules leads inevitably to the generation of interfaces. However, interfaces are always the boundary of two subsystems. In this case, the generated communication code on one side and the manually implemented code of the software modules on the other side. Mixing the module’s functional source code directly into the generated code has its disadvantages because it would mean to mix generated and manually written source code. If the model is changed, the source file is regenerated and the functional code has to be manually added again. This would reduce the benefits from MDSD dramatically.

The generation gap pattern solves this problem by providing a solution to combine manually implemented and generated code. The idea is to generate the source code in one class and perform customizations in another. The class for the manual adjustments inherits from the generated one to benefit from the generation. The automatically created classes are stored in a special folder, which should not be edited manually. However, an interface class for the module developer is generated during the first run and later ignored by the generator. This approach enables the addition of module code to the interface class without the need to create a new file.

To reduce the overhead for adding new modules, the generator creates also the communication code for the execution platform. If a module is executed, its parameters and inputs have to be loaded. Figure 7 shows how the generation gap pattern is combined with the decorator pattern to load and send data between system modules. If a module is triggered, the scheduler calls the execute method of the execution container. Its generated code receives the input data, unpacks it and provides it to the class containing the custom module code. After the delegated method in the custom class has finished its execution, the generated method packs the outgoing data and sends it to the succeeding modules.

This implementation differs from the usual decorator pattern. Usually, when using a decorator for extending a class, the extension class inherits from the base class. In this case, the base class is generated and the extensions are located in the manually written subclass. The delegation uses pure virtual methods to call the subclass.

Besides communication code, the configuration management is also generated. If a module is going to be initialized, the generated code calls a configuration manager, which parses a configuration file. The model contains the parameters. This has the benefit that a module developer does not need to take care of parameter definition or loading during initialization of his or her software module. The MDSD framework handles this by generating the source code for parsing a central

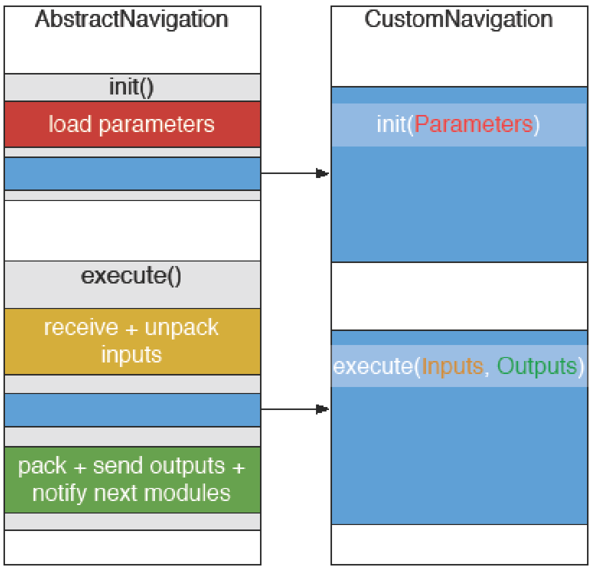

FIGURE 7 Combination of the decorator and the generation gap pattern: the generated class (left) calls methods in the customized class (right).

configuration file.

Because changes in the system design do not only affect the source code but may also change the documentation and infrastructure, it makes sense to also generate as many of these artefacts as possible.

To create documentation for the generated interfaces, we generate Latex files, which can then be used to create a PDF file. The generated documentation contains the description of all model elements as well as tables for in- and outgoing data, their types and the parameters of the module.

Because the generation gap pattern introduces a set of new classes, the build configuration gets more complex. To solve this problem, the generator creates also a build-system file for each module, which provides variables with the necessary include and source files.

Besides documentation and build-system files, the generator also creates unit tests. The integration of tests into the generator brings the benefit that errors caused by unsupported elements from UML or SysML are identified. For example, the unit test for the configuration loader would fail if a parameter type were either not supported by the code generator or by the underlying library.

With the definition of communication channels in the system model, it is possible to generate automatic tests that validate exchange of data between the module interfaces. Generated implementations of the modules send and validate received data. It is possible to specify valid and invalid data in the model. Furthermore, with the definition of all datatypes in the model, it is also possible to generate log file readers. File readers enable system tests based on comparison of the inputs' expected outcome with the navigation result. A continuous integration system deploys project changes on the target computer and executes the ATON software with stored input data. This way, tests validate that the software is working on the target system. In combination with continuous builds, it is possible to evaluate directly if project changes affect the system's results.

5. EVALUATION

ATON's navigation system uses optical sensors to estimate its current location and attitude. We model sensors, evaluation algorithms and a Kalman filter as software modules. Different models represent different scenarios and navigation software designs. A sensor simulation creates data for landing scenarios on the Moon. This scenario contains a camera, a star tracker, an IMU and a flash LiDAR. Besides crater navigation and feature tracking, a binary shadow matching supports this scenario.

To test the navigation system with real hardware, we performed flight tests on earth. The UAVs were equipped with two cameras, two altimeters, a star tracker emulator and an IMU. The resulting model contains 20 data types, 6 enumerations and 23 software modules each with up to 30 parameters.

The application of model-based approaches had a positive impact on the ATON project. With the generation of the interfaces and its data structures from templates the aimed goal of unification was reached. One template for all interfaces results in classes with the same structure, coding style and naming conventions. Furthermore, the central definition of data structures for all modules reduced misunderstandings in the development team and the need to convert the data within the interfaces.

Furthermore, the effort for the software development is reduced significantly. While changing the interfaces or data types was associated with several changes in the source code and documentation, it is now a simple task. After the change is applied to the model, the generator updates all related files. In particular, the complex communication and execution code needs to be implemented only once, and then, the template can be applied to all software modules and communication channels. This way, the development benefits twice: it has become less work to integrate new modules and in the end, changes either in the model or in one of the used libraries can be solved by doing the adjustments in only one place. For example, if the method to send messages between modules changes, it is sufficient to change the execution container's template. No manually written code needs to be updated.

Additionally, it is possible to create several configurations for different requirements. The system for a flight test on earth with an unmanned aerial vehicle needs different modules than a scenario simulating a landing on the Moon. To solve this challenge, it is possible to create different models, one with hardware drivers for flight tests and one with the sensor simulation for a lunar mission. This way, it is possible to use the same modules for different scenarios by only generating code from different models and using different configuration files.

In the course of the project, model changes became a regular development task. In the integration phases, small changes like updates of data types or parameters appeared on a weekly basis. The addition or removal of communication channels was rarer but its consequences on the source code were larger. Without the code generators, the developers would have to care about memory management, the event-driven execution of the tasks and data exchange between the software module's threads. Furthermore, design changes of the navigation system usually required adding or removing software modules. For example, the integration of a stereo chain, consisting of an epipolar geometry algorithm and stereo matching algorithms, would have required redeveloping the software completely. Customizations on that scale require changing the scheduling, data flow and configuration management. With MDSD, it is sufficient to just delete or add an element in the diagram. The generator creates all base classes and their integration into the scheduler and the communication system. Furthermore, generated log file writers stored all relevant data from flight tests. MDSD enabled us to create test setups, where generated log file readers replace sensors. This way, with enabling a build system option, the software can replay stored flight data instead of processing current sensor values. This approach enables to test the navigation system with real flight data, without available hardware sensors.

Nevertheless, the development team did not only benefit from code and document generation, the communication between the engineers of the different domains also improved. Before, interfaces and its documentation could be changed without realizing the potential impact to the rest of the system. With the model, changes are directly evaluated which, for example, immediately reveals incompatible types of communication channels. Thus, if types need to be changed, the affected interfaces are updated and the developers are immediately aware of the changes.

While the general efficiency increased by applying the model-based approaches, there were also some drawbacks. Even with the definition of a profile to customize the modeling language, the learning effort for UML and SysML was high. In addition to that, it is still possible to add elements to the model, which are not supported by the generator. With the high number of elements provided by UML and SysML, it is difficult to decide which one can be used in the given context. Moreover, even with the creation of collections of supported and customized elements in the editor, it is still possible to add elements of the base languages. The extension mechanism of UML is well equipped in adding parameters and constraints to the language, but the actual meta-model and thereby its native elements cannot be changed.

This restriction allows using available diagram editors for customized languages without adjustments, but the development of the code generator becomes more complex. One has to make sure that the code generator does not run into undefined conditions; all possible model constellations need to be covered. This validation code may easily exceed the generator code when using complex languages such as UML and SysML.

In addition to that, the generator needs a lot of logic to transform the generic language elements to the system's domain. Because the customized elements are represented by UML/SysML base elements, it is necessary to check if the elements have an extension and how to access them. Since it is not possible to modify the diagrams to set some domain relations automatically, the generator needs reference and identity checks. While two ports, linked by a connector element, represent a communication, it is not possible to access the opposite port by traversing the link. The additional logic required to resolve diagram relations reduces the maintainability of the generator code. It is relatively easy to change the actual templates, whereas the integration of new element types is more complex. E.g., the integration of a new diagram type for a special kind of event is challenging because it has to be integrated into the generator logic.

The project ATON has recently concluded a closed-loop test campaign on an unmanned aerial vehicle (UAV) with artificial craters targets on the ground. The software was able to navigate the UAV along a trajectory of 200 meters length. The position error was less than one meter in all dimensions. The next project step is to optimize the system for space missions.

6. CONCLUSION AND OUTLOOK

In this paper, we presented the introduction of model-based techniques into the development of a complex software system for real-time optical navigation for spacecraft. The project benefits from the application of model-based approaches. The efficiency has improved significantly through the introduction of modeling and code generation. While earlier, the integration of new modules has taken several days, it can now be achieved in a few hours. Because of this, the general project flexibility has increased. The reduced overhead for the integration of new modules lowered the inhibition threshold for system design changes. It is more likely to test the outcome of a new module, if the integration only takes a few hours instead of weeks. In addition, the number of errors decreased because of the unified interfaces and data types.

While the code generation and modeling in general turned out to be very efficient, the modeling language should be improved. The combination of UML and SysML covers all aspects of the system that are needed, but it requires a high learning effort for the system modelers. Furthermore, the languages are too powerful to be completely covered by the code generator. A solution is the development of a graphical domain-specific language from scratch [24]. Such a DSL is easy to learn and contains only elements that are supported by the code generator. Furthermore, the generator needs less logic, because it is not necessary to transform the model contents into the domain of the execution platform. The future work will be to apply such a domain-specific language to similar complex systems.

REFERENCES

[1] N. Ammann and F. Andert, "Visual Navigation for Autonomous , Precise and Safe Landing on Celestial Bodies using Unscented Kalman Filtering," in *Aerospace Conference, IEEE*, 2017.

[2] F. Andert, N. Ammann, and B. Maass, "Lidar-Aided Camera Feature Tracking and Visual SLAM for Spacecraft Low-Orbit Navigation and Planetary Landing," in *Advances in Aerospace Guidance, Navigation and Control*, Springer, 2015, pp. 605–623.

[3] H. Kaufmann, M. Lingenauber, T. Bodenmüller, and M. Suppa, "Shadow-Based Matching for Precise and Robust Absolute Self-Localization during Lunar Landings," in *Aerospace Conference, IEEE*, 2015.

[4] B. Maass, H. Krüger, and S. Theil, "An Edge-Free, Scale-, Pose- and Illumination-Invariant Approach to Crater Detection for Spacecraft Navigation," in *Image and Signal Processing and Analysis (ISPA)*, 2011.

[5] K. Vipavetz, T. A. Shull, and J. Price, "Interface Management for a NASA Flight Project using Model-Based Systems Engineering (MBSE)," *INCOSE Int. Symp.*, vol. 26, no. 1, pp. 1129–1144, 2016.

[6] J. Bézivin, "In Search of a Basic Principle for Model Driven Engineering," *UPGRADE*, vol. 5, no. 2, pp. 21–24, 2004.

[7] M. Perrotin, E. Conquet, J. Delange, and A. Schiele, "TASTE : A real-time software engineering tool-chain Overview , status , and future," in *Integrating System and Software Modeling (SDL)*, 2011, pp. 26–37.

[8] D. Kaslow, L. Anderson, S. Asundi, B. Ayres, C. Iwata, B. Shiotani, and R. Thompson, "Developing a CubeSat Model-Based System Engineering (MBSE) Reference Model - Interim status," in *Aerospace Conference, IEEE*, 2015.

[9] T. Bouabana-Tebibel, S. H. Rubin, and M. Bennama, "Formal modeling with SysML," in *Information Reuse and Integration (IRI), IEEE*, 2012, pp. 340–347.

[10] M. Mura, L. G. Murillo, and M. Prevostini, "Model-based Design Space Exploration for RTES with SysML and MARTE," in *Forum on Specification, Verification and Design Languages (FDL)*, 2008, pp. 203–208.

[11] B. Cole, G. Dubos, P. Banazadeh, J. Reh, K. Case, Y. F. Wang, S. Jones, and F. Picha, "Domain-Specific Languages and Diagram Customization for a Concurrent Engineering Environment," in *Aerospace Conference, IEEE*, 2013.

[12] C. Rupp and S. Queins, *UML2 Glasklar*, 3. Auflage. Carl Hanser Verlag GmbH & Co. KG, 2012.

[13] M. Iqbal, M. U. Khan, and M. Sher, "System Analysis and Modeling Using SysML," in Kim K., Chung KY. (eds) *IT Convergence and Security 2012. Lecture Notes in Electrical Engineering*, vol. 215, Springer, Dordrecht, 2013, pp. 1211–1220.

[14] C. Atkinson and T. Kühne, "Model-Driven Development: A Metamodeling Foundation," *IEEE Softw.*, vol. 20, no. 5, pp. 36–41, 2003.

[15] ISO/IEC 19509:2014: "Information technology - Object Management Group XML Metadata Interchange (XMI)," International Organization for Standardization, Geneva, Switzerland 2014.

[16] G. J. Holzmann, "Landing a Spacecraft on Mars," *IEEE Softw.*, vol. 30, no. 2, pp. 83–86, 2013.

[17] A. W. Brown, "Model-Driven Architecture," in *Software and Systems Modeling*, vol. 3 no. 4, Springer Verlag, 2004, pp. 314–327.

[18] O. Maibaum, D. Lüdtke, and A. Gerndt, "Tasking Framework: Parallelization of Computations in Onboard Control Systems," in *Betriebssysteme für zukünftige Rechnerarchitekturen. Autumn Meeting: Special Interest Group Operating Systems (ITG/GI)*, 2013.

[19] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley Professional, 2008.

[20] S. Gerard, C. Dumoulin, P. Tessier, and B. Selic, "Papyrus: A UML2 Tool for Domain-Specific Language Modeling," in *Model-Based Engineering of Embedded Real-Time Systems*, Springer Verlag, 2010, pp. 361–368.

[21] S. Efftinge and S. Zarnekow, "Extending Java - Xtend a New Language for Java Developers," 2011. [Online]. Available: https://pragprog.com/magazines/2011-12/extending-java. [Accessed: 08-Sep-2016].

[22] E. Gamma, R. Helm, J. Ralph, and J. Vlissides, "Structural Patterns," in *Design Patterns – Elements of Reusable Object-Oriented Software*, Edition 1., Addison-Wesley Professional, 1994, pp. 196–208.

[23] M. Fowler, "Generation Gap," in *Domain-Specific Languages*, Addison-Wesley Professional, 2010, pp. 571–573. ISBN: 0321712943

[24] T. Franz, "Entwicklung einer grafischen Modellierungssprache für ein ereignisgesteuertes Echtzeit-Laufzeitsystem," Bachelor Thesis, Baden-Württemberg Cooperative State University - Information Technology, 2015.

[1] Extension to the original DLRK conference publication: T. Franz, D. Lüdtke, O. Maibaum, A. Gerndt, "Model-Based Software Engineering for an Optical Navigation System For Spacecraft", in *65. Deutscher Luft- und Raumfahrtkongress (DLRK),* Braunschweig, Germany, 2016