TASKING MODELING LANGUAGE: A TOOLSET FOR MODEL-BASED ENGINEERING OF DATA-DRIVEN SOFTWARE SYSTEMS

Tobias Franz[1], **Ayush Mani Nepal**[1], **Zain A. H. Hammadeh**[1], **Olaf Maibaum**[1], **Andreas Gerndt**[1,2], **and Daniel Lüdtke**[1]

[1]*German Aerospace Center (DLR), Institute for Software Technology, 38108 Braunschweig, Germany*
[2]*Center for Industrial Mathematics (ZeTeM), University of Bremen, 28359 Bremen, Germany*

ABSTRACT

The interdisciplinary process of space systems engineering poses challenges for the development of the on-board software. The software integrates components from different domains and organizations and has to fulfill requirements, such as robustness, reliability, and real-time capability. Model-based methods not only help to give a comprehensive overview, but also improve productivity by allowing artifacts to be generated from the model automatically. However, general-purpose modeling languages, such as the Systems Modeling Language (SysML), are not always adequate because of their ambiguity resulting from their generic nature. Furthermore, sensor data handling, analysis, and processing of data in on-board software requires focus on the system's data flow and event mechanism. To achieve this, we developed the Tasking Modeling Language (TML) which allows system engineers to model complex event-driven software systems in a simplified way and to generate software from the model. Type and consistency checks on the formal level help to reduce errors early in the engineering process. TML is focused on data-driven systems and its models are designed to be extended and customized to specific mission requirements. This paper describes the architecture of TML in detail, explains the base technology, the methodology, and the developed domain specific languages (DSLs). It evaluates the design approach of the software via a case study and presents advantages as well as challenges faced.

1. INTRODUCTION

Space missions are experiencing a transformation on how their operations are conducted. Classically, experiments were executed by following a static, pre-flight designed list of commands. Now, on-board computers analyze sensor data and react according to the current situation. This way, a system's on-board software can handle unexpected situations and act on cases where data was not yet available before launch. However, this kind of automatic planning poses new requirements for on-board data processing. Large amounts of sensor data need to be processed in real-time. In addition to that, some applications, such as image processing, require high computational power on board.

Developing end-to-end data processing pipelines in one system is challenging as components from different providers have to be integrated and connected. Furthermore, design changes have to be incorporated into source code, documentation, and also into the build configuration. Model-driven software development (MDSD) supports this process by generating communication and interface code from a central data model [1]. Engineers can apply design changes to the model and regenerate updated artifacts from it. The model can also be used to validate the developed design and acts as a communication method within the team. However, in data-/event-driven systems, general purpose modeling languages are not suited to MDSD activities [1]. To properly configure when components are executed, the model language needs to incorporate domain-specific aspects.

Some on-board software, such as attitude and orbit control system (AOCS), and the majority of on-board data processing applications can be implemented as a network of data-driven computational nodes. In such an implementation, a computational node starts execution when input data are available and does not need to explicitly wait for the predecessor computational nodes to finish. Also, computational nodes exchange data via data containers which represent interfaces between them. The computational nodes are known as tasks and data containers are referred to as channels, hence, the model is known as task-channel model. This model has potential for modularity, enabling reusability of developed applications. The Tasking Framework [2] is an open-source[1] C++ library to develop on-board software using the task-channel model. It provides abstract classes with virtual methods following the event-driven programming paradigm. With Tasking Framework, an application is realized by a directed graph of connected tasks and channels. To generate source code for an application based on Tasking Framework, a model for MDSD needs to depict this graph.

In this paper, we present such a modeling language for event-/data-driven software systems, implemented based on the open source tool Virtual Satellite[2]. Virtual Satellite aims to model systems throughout their complete life-

[1]https://github.com/DLR-SC/tasking-framework
[2]https://github.com/virtualsatellite/

cycle [3]. Incorporating early-phase system design models into the MDSD methodology allows reuse of existing system model artifacts. For the model-driven development, this further improves productivity and maintainability.

Remainder of this paper is structured as follows: the next section introduces relevant literature, Section 3 presents our concept and implementation, and Section 4 evaluates our modeling approach by applying it to a example project from the space domain. Finally, Section 5 concludes the paper with a future outlook.

2. BACKGROUND AND RELATED WORK

In this section, we introduce basics for on-board data processing and model-driven software engineering.

2.1. On-Board Data Processing

Aerospace projects have started to include more autonomous systems or subsystems to carry out missions, e.g., on distant celestial bodies. Such systems require more on-board data processing, which demands high performance computing resources. Multi-core platforms are promising to fulfill these computational requirements [4] as they provide high performance with low power consumption in comparison to high frequency uniprocessors. However, it is often difficult to write applications that execute in parallel. This adds more complexity to the design and implementation processes of on-board data processing applications.

Software that process data onboard can be modeled as a directed data-flow graph, where vertices represent processing nodes and data are forwarded to the next node in the pipeline. Processing nodes work concurrently. They do not have to wait for preceding nodes to finish their job. They can start executing as soon as their first input data is available. Therefore, these applications have a high potential for modularity and parallelism. Implementing concurrency in an application mitigates the migration to multi-core platforms. And, exploiting the modularity improves the re-usability of the developed applications. Such a data-driven approach has been used to develop on-board software that executes and controls complex experiments on board the International Space Station (ISS) [5].

In some aerospace missions, sensor data are processed on board during critical operations, such as autonomous take-off and landing [6]. To ensure the safety and reliability of the mission, on-board data processing software should be real-time capable and may need to guarantee the availability of needed data on time. For example, during the autonomous landing operation [6] or on-board rendezvous navigation [7]. For such safety critical applications, engineers need to perform timing analysis of the software and ensure that end-to-end real-time constraints are met. Timing analysis should also cope with the aforementioned software complexity caused due to parallelism [8]. Consequently, increasing on-board data processing in current and future space missions introduces more challenges in the development of onboard software.

2.2. Model-Driven Engineering

The European Space Agency (ESA) has developed a tool chain for MDSD called The ASSERT Set of Tools for Engineering (TASTE) [9]. TASTE focuses on the error-prone process of integrating different software components. It uses a textual language (ASN.1) [3] to define data types and a graphical language (AADL) to model the system architecture [9]. The TASTE generator not only generates source code, but also produces build configuration files for the system.

ESA, furthermore, has developed a reference architecture for space systems [10]. It contains a modeling language, namely, "Space Component Model" [11] to specify different components of the system. The reference architecture does not impose any specific tool, but provides a prototype implementation based on the Eclipse Modeling Framework (EMF)[4]. The reference architecture's modeling environment is based on several domain specific languages (DSLs) and also provides code generation capabilities [11].

NASA Jet Propulsion Laboratory (JPL) investigated methodologies to reduce the learning overhead of SysML [12]. They suggest to build DSLs based on SysML and to add domain-specific notations to the model. The generic nature of the Unified Modeling Language (UML) or SysML allows users to model various different kinds of systems [1]. However, this generic nature of these modeling languages not only increases the learning effort for users but it also adds difficulties in the model-driven development (MDD) process with code generation [1]. Gray and Rumpe [13] also observed difficulties in using UML and SysML with domain experts that are not familiar with modeling in their daily work. They suggest investigating whether it is more adequate to design a DSL from scratch rather than customizing an existing general purpose language.

In a case study, Visser [14] investigated domain-specific engineering. He collected guidelines and patterns on how DSLs can be implemented. Before developing a DSL, he suggests to do a domain analysis and to investigate which aspects of the system have to be modeled. Atkinson and Kühne [15] describe different forms of meta-modeling for MDD. They also defined a set of technical requirements that MDD techniques should support. They argued that besides available concepts for modeling, MDD infrastructure has to define notations to depict models,

[3]https://www.itu.int/rec/T-REC-X.680/
[4]https://www.eclipse.org/modeling/emf/

specify how the model entities represent the real-world elements, and also provide concepts to facilitate user extensions.

3. TASKING MODELING LANGUAGE

As highlighted in Section 2.1, developing software for on-board data processing poses new challenges. This section analyses requirements for a model-based solution to these challenges and presents the design of our modeling environment.

3.1. Domain Analysis

As shown in Figure 1, on-board data processing is highly data-driven. Sensors produce data, which is then analyzed by the processing components, then actuators are controlled accordingly. The development of such systems faces some challenges: firstly, the system components (sensors, processing components, actuators) are usually developed by different institutions or obtained from external suppliers. Thus, software for these components might follow different patterns, guidelines, or coding styles. However, they have to be integrated together for the whole system to run. Secondly, components themselves and their interactions are subject to frequent changes during the course of the development. This might yield changes in the interface description of these components. Furthermore, the processing pipeline might need adaptation, for example, to incorporate additional components or to remove the deleted ones. All of these changes need to be reflected in multiple places, for instance, in the interface documentation, in the software architecture and subsequent components. Model-based approaches can alleviate these challenges. As the central point of truth, the model takes all modifications and generates required artifacts such as documents and source code. Moreover, component developers are notified about the respective changes. This helps to eliminate inconsistencies in the development process and helps to resolve misunderstanding between the development team members caused by the differences of perspective. However, to benefit from modeling through improved productivity and communication within data-driven projects, models have to focus on certain aspects. We elaborate on these aspects in the following paragraphs.

System components consists of processing software that creates, analyzes, or handles data. Its interface descriptions have to be standardized and specified in an unambiguous way. This allows engineers to incorporate devices from different research institutions and commercial vendors. Additionally, the software should be as modular as possible so that adding new elements or removing the existing ones becomes easier. Furthermore, causality of components has to be defined clearly to reduce side effects due to any change.

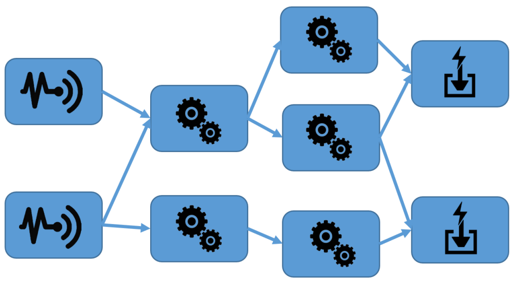


Figure 1. Software for on-board data processing usually contains software components for sensors, processing components and actuators.

Data flow is the central aspect of the on-board data processing software. It describes how components are connected and how they communicate. Important are again coherent interface descriptions. For a proper data flow, related data specifications are of specific relevance. They do not just define type and range of valid data but may contain specific information as well, e.g., data bandwidth.

Data input events should be configurable to trigger components once all necessary input data are available. This way, the processing pipeline of the on-board software can handle data as fast as possible.

Extensibility is needed to consider project-specific requirements. As a modeling language for on-board software, it is tailored to a specific domain. However, different on-board software solutions require individual details in the model. E.g. a project-specific network implementation could require some details in the model that has to be specified for each channel. A modeling language that aims to be used in multiple projects needs to be customizable to specific requirements.

As a result, we pose the following requirements to the modeling environment for on-board data processing:

R.1 The environment must allow the specification of component interfaces in a data model

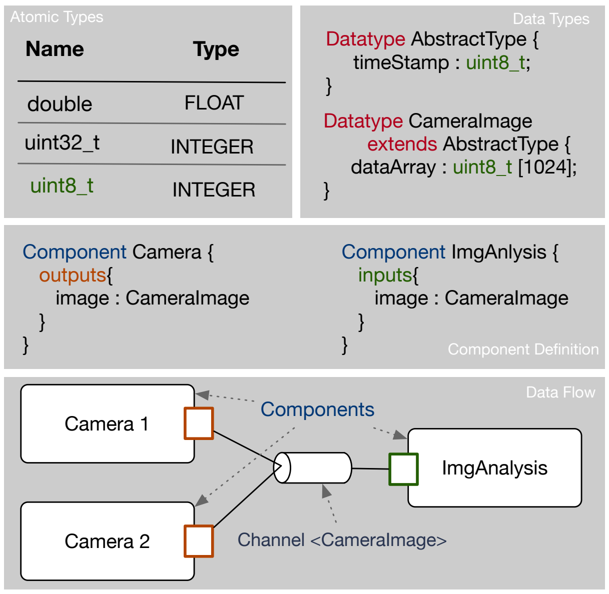
R.2 The modeling language must allow the specification of the data flow.

R.3 Execution event handling shall be supported.

R.4 The environment shall be extensible to specific project requirements.

3.2. Concept

A model is an abstraction of a complex system with focus on the relevant aspects for a specialized purpose. To reduce the learning overhead of the modeling language, it should be specialized to its purpose as much as possible. To model the data flow and event mechanism of the data processing software, we decided to develop a DSL.



Atomic Types

Name	Type
double	FLOAT
uint32_t	INTEGER
uint8_t	INTEGER

Data Types

```
Datatype AbstractType {
    timeStamp : uint8_t;
}
Datatype CameraImage
      extends AbstractType {
    dataArray : uint8_t [1024];
}
```

Component Definition

```
Component Camera {
   outputs{
     image : CameraImage
   }
}
```

```
Component ImgAnlysis {
   inputs{
     image : CameraImage
   }
}
```

Figure 2. Different types of model representation: Atomic types are specified in a list, data types and components in a textual DSL and the data flow in a graphical diagram.

In contrast to general purpose languages, such as UML or SysML, a DSL only contains aspects that are relevant to the modeling purpose.

Model notation. Developing a DSL allows one to specify a concrete syntax. Different aspects of the model can be rendered individually in an appropriate way. Modeling interfaces and the system data flow require specification of applicable data types first. As shown in Figure 2, a simple way to define data types is to specify required atomic types in a list, and then, using them to create composite types using a textual DSL. Similar to programming languages, textual DSLs can also have keywords to specify properties, such as, multiplicity, inheritance, etc. In combination with editors featuring autocompletion, textual DSLs are relatively straightforward to use. They provide a clear and flexible interface for users to define model elements, especially, when a graphical layout is not necessary. For the simplicity and clarity, besides data-type definitions, the specification of component interfaces are also done using a textual language.

The data flow connects the different software components and, thereby, provides a good system overview. As shown in Figure 2, to facilitate and support this overview, the data flow is modeled in a graphical diagram. Here, components are represented as nodes; its inputs and outputs are shown as ports on these nodes. It is, then, possible to connect these components via edges. To model n:m connections and to specify the storage/data interface of these links, users can add channels to the diagram.

The different languages for specification of data types, component interfaces, and data flow are part of a common data model. Each language represents a view on this shared model. For instance, data types are specified in one language and used in others. Editors for the different views integrate changes into the common model. This way, the views and editors are synchronized. For example, if a component has two inputs and one output, its instances in the diagram will automatically have the respective number of ports.

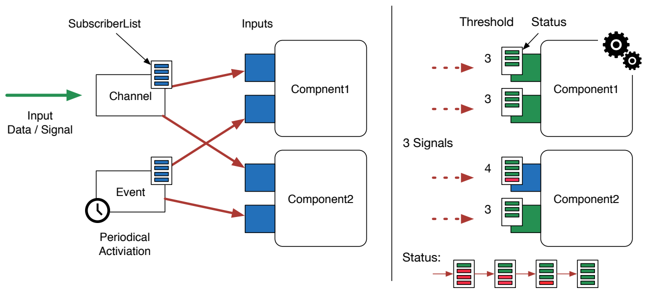


Figure 3. Components are executed event-based. Component inputs subscribe to update events of data channels or other event sources. Once the defined threshold of signals (e.g. number of data items) is reached, the input gets activated. When all inputs are active, the component gets executed.

Event handling. Components need to be executed once the required input data are available. The configuration of data events and the resulting generated code are based on the Tasking Framework. As shown in Figure 3, components subscribe to their input channels. This way, they receive notifications about when data are available. A threshold number per input defines how many data items are required for an execution of the component. Once this threshold is reached, the corresponding input is activated. When all inputs are activated, the component gets executed. Input data can also be marked as final or optional: Final inputs automatically execute the component if activated, regardless of the availability of other data. Optional inputs are not required for an execution of the component. Besides data events, it is also possible to trigger components with bare signals, such as periodic timers.

Model checking. System models enable to find issues and design flaws early in the engineering process. The Tasking Modeling Language (TML) validates that communication channels have compatible interfaces, parameters are filled with correct value types, and units are set correctly. Furthermore, users are notified if components are not connected properly.

Artifact generation. Generators create source code, documentation, and configuration files from the model. The generation is based on templates with place-holders that are filled with data from the model. To improve the integration of software components, the source code generator creates interface classes and the communication

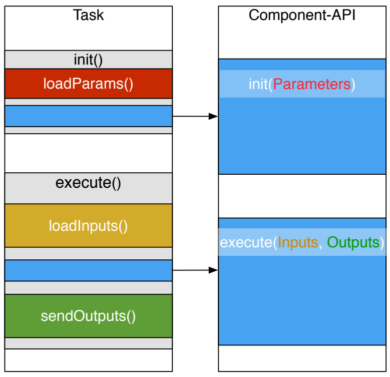


Figure 4. Generated functionality is added into abstract classes, manual customization is implemented in derived classes. If the model is changed, abstract classes are re-generated; the derived classes are not.

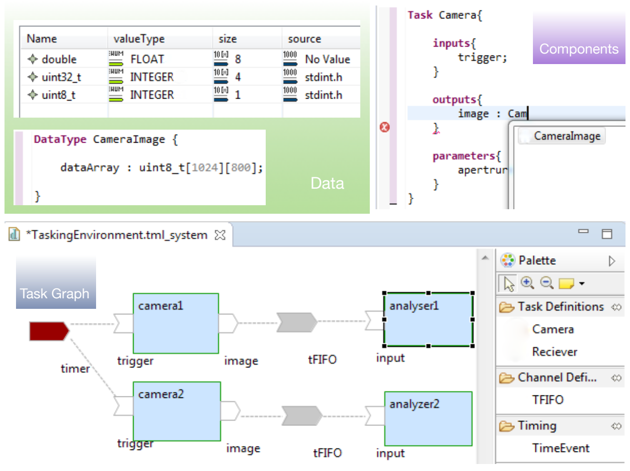
code. This way, if new elements have to be integrated into the processing pipeline, generated source code automatically re-routes the data.

The implementation of multi-threaded source code for on-board real-time applications can be challenging. To simplify the software development, the modeling methodology provides a simplified user front-end for the Tasking Framework. Users should be able to create data- and event-driven on-board software without being an expert in multi-threaded programming. To achieve this, TML abstracts from the complexity of the executable source code. Users only see details that are necessary to create and configure the data flow and the event mechanism of the system. As shown in Figure 4, the code generator prepares incoming data and parameters, and provides a simplified API to components. To enable customization of the generated source code, the generator implements the generation gap pattern [16]. This pattern separates generated logic from the custom code using the inheritance mechanism. Abstract classes are re-generated whenever the model is updated, whereas, the user modified code in derived classes is not over-written by the generator.

Extensibility. Extensibility of TML is implemented on multiple levels. First, the model itself can be extended with new elements. Therefore, we use the meta-modeling pattern of *Promotion* [17]. Channel and component types, dynamically defined in the model, can be instantiated in the component diagram. The second level is to customize the code generator by adapting the code templates to specific project-needs. As already mentioned before, the generation gap pattern enables developers to further customize the generated code.

3.3. Implementation

TML is built as an extension of Virtual Satellite, which is an extensible model-based systems engineering (MBSE)



Name	valueType	size	source
double	FLOAT	8	No Value
uint32_t	INTEGER	4	stdint.h
uint8_t	INTEGER	1	stdint.h

```
DataType CameraImage {

    dataArray : uint8_t[1024][800];

}
```

```
Task Camera{

    inputs{
        trigger;
    }

    outputs{
        image : Cam
    }

    parameters{
        apertrur
    }
}
```

Figure 5. Software for on-board data processing usually contains software components for sensors, processing components and for actuators.

tool developed and maintained by DLR [3]. Integration of TML in such a systems engineering tool allows engineers to reuse already modeled artifacts from early-phase design specifications.

Virtual Satellite is build on top of the Eclipse modeling environment. It assists software engineering by using the MDSD process to generate user interface (UI) snippets and other software artifacts automatically from the model. The EMF framework provides code generators for producing Java interface and implementation classes for all model entities and their editors. Besides that, the Virtual Satellite layer provides further customization of the model editors, UI snippets, and improves the model validation and verification capabilities. With the help of model validators, Virtual Satellite assists to improve the overall quality of the software product. Furthermore, it also supports backward compatibility of the model by providing migrators.

Attributes of TML model elements, such as the name, can be modified using generated UI editors. Furthermore, as shown in Figure 5, TML provides customized Xtext[5] editors for specifying data-types, components, and channel definitions. Keywords, auto-formatting, and syntax highlighting of the DSL editors improve user experience. The diagram editor for the data flow is implemented with the Graphiti[6] framework in Eclipse. Graphiti provides a simple API to create and synchronize model domain objects with their diagram representations. Via drag and drop, users can instantiate tasks, channels, or time-events and connect them together using links. Furthermore, a double-click on any diagram element opens the respective UI editor, where all of its attributes can be modified.

[5]https://www.eclipse.org/Xtext/
[6]https://www.eclipse.org/graphiti/

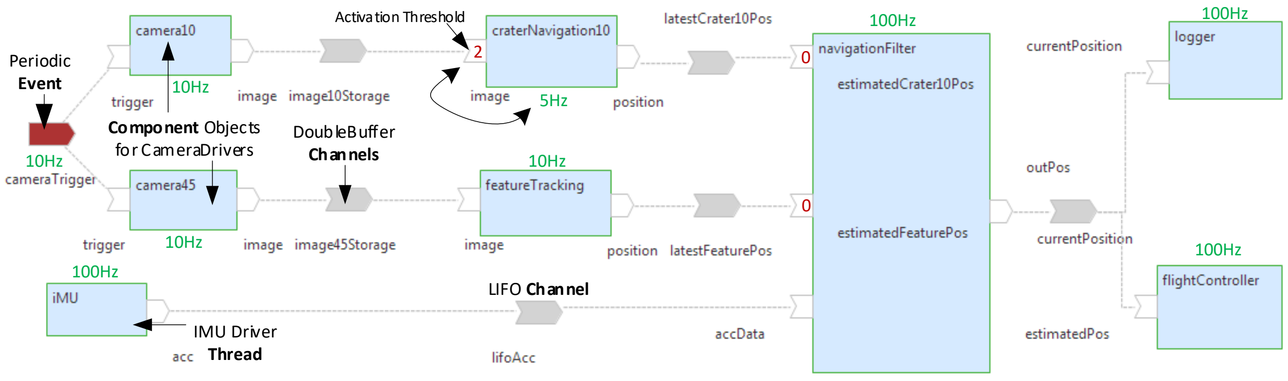


Figure 6. TML component diagram showing the architecture for the optical navigation system of the project ATON [1]. Cameras are triggered periodically and provide input for other processing components. A navigation filter fuses all processing results and creates an estimated position which is logged and sent to the flight controller.

4. CASE STUDY

To evaluate the modeling capabilities of TML, we re-implemented the SysML model of the project ATON (Autonomous Terrain-based Optical Navigation) using TML [1]. Figure 6 shows the component diagram of this implementation. The general architecture is data/event-driven, i.e., processing components (also referred to as tasks) can be executed as soon as their required input data is available, alternatively, they can also be triggered by periodic events. To control the frequency of a task, user can specify an activation threshold to its input. The activation threshold specifies how many input samples are required to trigger the task. For instance, cameras shown in Figure 6 are triggered by a periodic event of 10 Hz and their output images are consumed by two tasks. In the example use case, the crater navigator is only executed at the rate of 5 Hertz (Hz) as it is only activated by every second image. The inertial measurement unit (IMU) runs as a non-Tasking framework thread at the rate of 100 Hz. The activation threshold of "0" on two navigator filter inputs marks them as optional. Optional input data are used if they are available, i.e., tasks can be executed without them. As a result, frequency of the navigation filter and the IMU in the given example becomes equal, because, every data produced by the IMU will trigger the navigation filter.

4.1. Evaluation of the Requirements

The data flow diagram in Figure 6 shows how requirement R.1 and R.2 are fulfilled. The data flow is presented in the graphical diagram, showing all component instances and their connections. Furthermore, component types are represented in a textual language. The event handling (Requirement R.3) is implemented through the trigger components and input activation mechanism.

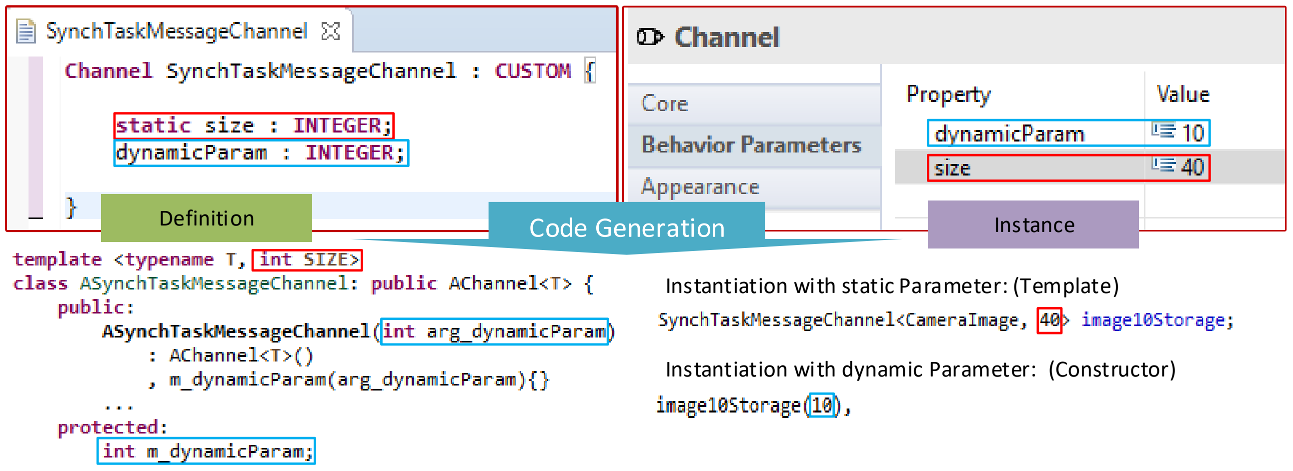
The case study also demonstrates how extensibility, as demanded by Requirement R.4, is achieved. The use case requires a customized data storage implementation, which also needs to be configured from the model. Figure 7 shows how the new channel type `SynchTaskMessageChannel` is modeled. The channel along with its parameters is instantiated in the component diagram (upper-right part of Figure 7). The TML generator creates base classes for these kinds of type extensions and instantiates them in the object model as specified in the component diagram. Parameter values, as specified in the model, are added to the generated base classes (bottom-left part of Figure 7). And their value is added in the object instances (bottom-right part of Figure 7).

As a result, the implementation of MDSD meets all our requirements.

4.2. Comparison with the SysML Solution

The TML was developed as a successor of the predecessor based on SysML [1]. Even though the former approach increased productivity of the development process, a conclusion of the work was that the modeling language should be improved. TML represents the next iteration of the modeling language. It is explicitly designed for the purpose of data-/event-driven systems. A component diagram in UML or a block diagram in SysML, both allow more than 40 different elements to be added. In contrast to that, the TML component diagram allows only four basic components (triggers, tasks, channels, and connections). Inputs and outputs of the different processing units are incorporated automatically. Thereby, TML helps to maintain consistency and removes ambiguity posed by the general-purpose modeling languages.

The concise modeling approach of TML has significant influence on the code generator as well. This is depicted in Figure 8. Unlike the one in SysML, TML generator does not have to transform the general-purpose modeling elements into the on-board data processing domain. Furthermore, as all model elements in TML have a direct



```
SynchTaskMessageChannel

Channel SynchTaskMessageChannel : CUSTOM {

    static size : INTEGER;
    dynamicParam : INTEGER;

}
```

Property	Value
dynamicParam	10
size	40

```
template <typename T, int SIZE>
class ASynchTaskMessageChannel: public AChannel<T> {
    public:
        ASynchTaskMessageChannel(int arg_dynamicParam)
            : AChannel<T>()
            , m_dynamicParam(arg_dynamicParam){}
        ...
    protected:
        int m_dynamicParam;
```

Instantiation with static Parameter: (Template)

```
SynchTaskMessageChannel<CameraImage, 40> image10Storage;
```

Instantiation with dynamic Parameter: (Constructor)

```
image10Storage(10),
```

Figure 7. The channel type extension mechanism. Channel types can define parameters that are available in its model instance and also in the generated code. The implementation type 'CUSTOM' creates base classes for its implementation and instantiates this class in the system's object model.

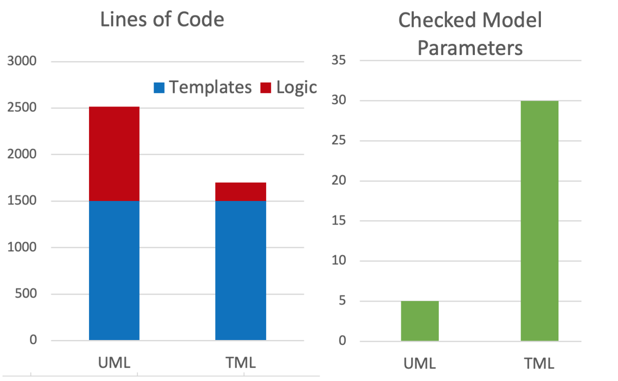


Figure 8. Comparison of the code generator with UML/SysML and TML model. Generation from the TML model needs less mapping and transformation logic. It furthermore contains more domain-specific validation of properties.

mapping to the generated source code, their parameters have clearly defined constraints, which are verified and validated directly in the model.

5. CONCLUSION AND FUTURE WORK

Sensor data handling, analysis, and processing of data in on-board software requires consideration of the system's data flow and event mechanism. This work presents a language for event and data-driven software systems. TML incorporates a set of textual and graphical DSLs, which allow system engineers to model complex event-driven software systems. The textual DSLs in TML provide a strong-typed syntax to define data types and system components, whereas a graphical DSL enables users to design the composition and data flow of the system. The TML environment facilitates MDSD by incorporating a code generator which generates executable C++ source-code, unit tests, its build configuration and documentation. This way, the model acts as a single point of truth, and changes on the data flow can be handled by re-generation. By default, the code generator utilizes the Tasking Framework, an open-source C++ software execution platform developed by DLR. This allows software modules to run concurrently in separate tasks, exchange data between them via channels, and to schedule task-execution based on events. TML is focused on even-driven systems, nevertheless, its infrastructure and models are designed to be extended and customized to specific mission requirements. It provides three levels of extension mechanism: firstly, new components, such as channel types and parameters, can be added to the model dynamically. Secondly, the code generator can be customized by extending or implementing new code templates. This makes it possible for TML to support new execution environments and even new programming languages. Furthermore, the generated code is designed to be extended by manually written code. This combination of a modeling language, customized to data-flow oriented systems with a highly extensible artifact generation, enables effective support for the development of on-board software.

Future work will focus on extending formal verification of TML models. Goal is to search for potential dead- and live-locks, and to analyze and improve the data flow of the system. Projects building up on TML are investigating reconfiguration planning of an on-board software modeled using TML [18]. TML will also evolve and improve by facilitating new features of the DLR Tasking Framework.

REFERENCES

[1] Tobias Franz, Daniel Lüdtke, Olaf Maibaum, and Andreas Gerndt. Model-based software engineer-

ing for an optical navigation system for spacecraft. *CEAS Space Journal*, 10(2):147–156, 2018.

[2] Zain Alabedin Haj Hammadeh, Tobias Franz, Olaf Maibaum, Andreas Gerndt, and Daniel Lüdtke. Event-driven multithreading execution platform for real-time on-board software systems. In *15th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*, pages 29–34, Juli 2019.

[3] Philipp. M. Fischer, Daniel. Lüdtke, Caroline. Lange, Frank. C-. Roshani, Frank. Dannemann, and Andreas. Gerndt. Implementing model-based system engineering for the whole lifecycle of a spacecraft. *CEAS Space Journal*, 9(3):351–365, 2017.

[4] Guillermo Ortega and Roger Jansson. GNC application cases needing multi-core processors. https://indico.esa.int/event/62/contributions/2787, October 2011. 5th ESA Workshop on Avionics Data, Control and Software Systems (ADCSS).

[5] Arnau Prat, Jan Sommer, Ayush Mani Nepal, Tobias Franz, Hauke Müntinga, Andreas Gerndt, and Daniel Lüdtke. The beccal experiment design and control software. In *Proceedings of the 2021 IEEE Aerospace Conference*, pages 1–9, 2021.

[6] Sergio Chiesa, Sara Cresto Aleina, Giovanni Antonio Di Meo, Roberta Fusaro, and Nicole Viola. Autonomous take-off and landing for unmanned aircraft system: Risk and safety analysis. In *29th Congress of the International Council of the Aeronautical Sciences*, September 2014.

[7] Eicke-Alexander Risse, Kurt Schwenk, Heike Benninghoff, and Florian Rems. Guidance, navigation and control for autonomous close-range-rendezvous. In *Deutscher Luft- und Raumfahrtkongress 2020*, October 2020.

[8] Xavier Palomo, Mikel Fernandez, Sylvain Girbal, Enrico Mezzetti, Jaume Abella, Francisco J. Cazorla, and Laurent Rioux. Tracing Hardware Monitors in the GR712RC Multicore Platform: Challenges and Lessons Learnt from a Space Case Study. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:25, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[9] Eric Conquet, Maxime Perrotin, Pierre Dissaux, Thanassis Tsiodras, and Jérôme Hugues. The taste toolset: turning human designed heterogeneous systems into computer built homogeneous software. In *European Congress on Embedded Real-Time Software (ERTS 2010)*, Toulouse, France, May 2010.

[10] Elena Alaña, Javier Herrero, Santiago Urueña, Krystyna Macioszek, and Daniel Silveira. A reference architecture for space systems. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–2, 2018.

[11] Marco Panunzio. Specification of the metamodel for the osra component model. *ESA, Tech. Rep.*, 2017.

[12] Bjorn Cole, Greg Dubos, Payam Banazadeh, Jonathan Reh, Kelley Case, Yeou-Fang Wang, Susan Jones, and Frank Picha. Domain-specific languages and diagram customization for a concurrent engineering environment. In *2013 IEEE Aerospace Conference*, pages 1–12. IEEE, 2013.

[13] Jeff Gray and Bernhard Rumpe. UML customization versus domain-specific languages. *Software & Systems Modeling*, 17(3):713–714, 2018.

[14] Eelco Visser. Webdsl: A case study in domain-specific language engineering. In *International summer school on generative and transformational techniques in software engineering*, pages 291–373. Springer, 2007.

[15] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.

[16] Martin Fowler. Generation gap. In *Domain-Specific Languages*, page 571–573. Addison-Wesley Signature, 2010.

[17] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–46, 2014.

[18] Andrii Kovalov, Tobias Franz, Hannes Watolla, Vishav Vishav, Andreas Gerndt, and Daniel Lüdtke. Model-based reconfiguration planning for a distributed on-board computer. In *Proceedings of the 12th System Analysis and Modelling Conference*, pages 55–62, 2020.