Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems

Conference Paper · July 2019

CITATIONS
0

READS
122

5 authors, including:

Zain Hammadeh
German Aerospace Center (DLR)
13 PUBLICATIONS **68** CITATIONS
SEE PROFILE

Tobias Franz
German Aerospace Center (DLR)
12 PUBLICATIONS **116** CITATIONS
SEE PROFILE

Daniel Lüdtke
German Aerospace Center (DLR)
60 PUBLICATIONS **293** CITATIONS
SEE PROFILE

Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems

Zain A. H. Hammadeh, Tobias Franz, Olaf Maibaum, Andreas Gerndt, Daniel Lüdtke
Simulation and Software Technology
German Aerospace Center (DLR)
Braunschweig, Germany
{zain.hajhammadeh, tobias.franz, olaf.maibaum, andreas.gerndt, daniel.luedtke}@dlr.de

***Abstract*—The high computational demand and the modularity of future space applications make the effort of developing multithreading reusable middlewares worthwhile. In this paper, we present a multihreading execution platform and a software development framework that consists of abstract classes with virtual methods. The presented work is written in C++ following the event-driven programming paradigm and based on the inverse of control programming principle. The platform is portable over different operating systems, e.g., Linux and RTEMS. This platform is supported with a modeling language to automatically generate the code from the given requirements. Our platform has been used in already flying satellites, e.g., Eu:CROPIS.**

We present in this paper an example that illustrates how to use the proposed platform in designing and implementing an on-board software system.

***Index Terms*—RTOS, Multithreading, Event-driven**

I. Introduction

Modern space applications demand high performance computing resources to carry out the increasing computational requirements of on-board data processing and sophisticated control algorithms. On the one hand, *multicore* platforms are promising to fulfill the computational requirements properly [1] as they provide high performance with low power consumption compared with high frequency uniprocessors. However, it is quite often not easy to write applications that execute in parallel. On the other hand, sensors are slow and cannot be on a par with the computing resources. Self-suspending processes are usually used to read from sensors, which makes timing more complicated and presents high pessimism and, thus, high over-provisioning.

In this paper, we present an event-driven multithreading execution platform, which is written in C++ following the inverse of control programming principle to improve reusability. We call our execution platform *Tasking Framework*. Tasking Framework provides abstract classes, which facilitates the implementation of space applications as event-driven task graphs. It also provides a multithreading execution based on POSIX, C++11 threading, and OUTPOST [2], which makes Tasking Framework compatible with Linux, RTEMS and many other real-time operating systems (RTOS).

Tasking Framework is motivated with lessons learned from the Bispectral Infra-Red Detection (BIRD) [3] attitude control system. The BIRD satellite launched in 2001. BIRD used a distributed Kalman filter [4] to estimate the attitude state vector of the satellite. This filter comprises several estimation and prediction modules executed by the controller thread. Each estimation module computes one value in the attitude state vector, for example, the sun vector from the sun sensor input values, the predicted sun vector and expected control effect from the last control cycle, a rate from the new sun vector or magnetic field vector, or the best rate by cross checking magnetic field vector rate, sun vector rate and measured rate from gyroscopes. The computation order is given by the data flow between the estimation modules. The order was given by a call sequence of the estimation modules in the controller thread.

During the development of the BIRD attitude control system some timing issues arose from the limited computing power of the on-board computer, and the timing requirements imposed by the sensors. BIRD used for all threads a predefined time slot in 500 ms cycle. The star tracker reported the attitude quaternion after 375 ms. The output buffers of the five actuators have to arm at 450 ms in the control cycle. By this, only 75 ms remain for all the computations inside the attitude control system. With the means of the event-driven paradigm, an as soon as possible scheduling of the computations is possible, which realizes timing constraints in the end of the control cycle. Only the computations that depend on the star tracker data have to be computed after 375 ms in the control cycle.

Tasking Framework has been used in the following projects: Autonomous Terrain-based Optical Navigation (ATON) [5], Euglena Combined Regenerative Organic food Production In Space (Eu:CROPIS) [6], Matter-Wave Interferometry in Weightlessness (MAIUS) [7], and Scalable On-Board Computing for Space Avionics (ScOSA) [8].

The rest of the paper is organized as follows: We present the basic concepts of Tasking Framework in Section II. In Section III, we elaborate the execution model. Our modeling language is presented in Section IV. Section V presents a case study of using Tasking Framework with the proposed Tasking Modeling Language (TML). After presenting our execution platform, we address the related work in Section VI, and we conclude in Section VII.

II. Tasking Framework

Embedded system applications are often described as a graph, which illustrates the software components and the de-

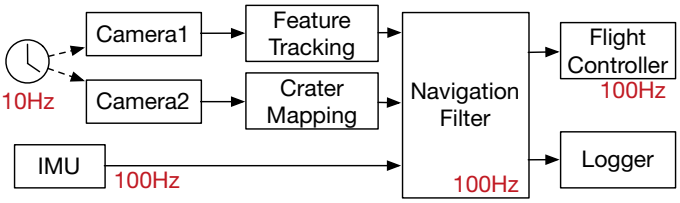

Fig. 1. Optical navigation system

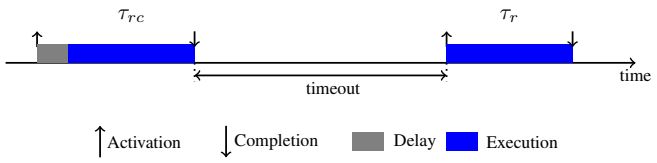

Fig. 2. An example of using the relative time. τ_{rc} sends a request command to the sensor. After the timeout occurrence, the following task τ_r reads the response sent by the sensor.

pendencies among them. Figure 1 shows an optical navigation system for spacecraft, which is a part of the ATON project [5]. Real-time capabilities are necessary to analyze optical sensor data and to react on the system's estimated position. The Tasking Framework is used to periodically trigger the cameras and to execute the image analyzer modules as soon as all required input data is available.

In this system, two cameras are triggered by a periodic timer and the images are then transferred to different analyzer components. The first one is a *feature tracking* component that estimates a relative movement, the second is a *crater navigation* component that tries to match craters on the Moon in the input images with a catalog of craters. The output of these components is then transmitted to the *navigation filter*. The navigation filter uses a Kalman filter to fuse the inputs with data from an additional inertial measurement unit (IMU) to get an estimated output position, which is then logged and sent to the flight controller. Tasking Framework is used in this example to integrate all these components.

Tasking Framework is implemented as a *namespace* `Tasking`, which comprises abstract classes with few virtual methods. It consists of the *execution platform* and the *application programming interface (API)*. Using the Tasking Framework, applications are implemented as a graph of *tasks* that are connected via *channels*, and each task has one or more *inputs*. Periodic tasks are connected to a source of *events* to trigger the task periodically, see Figure 7. In practice:

- Each computation block of a software component is realized by the class `Tasking::Task`. The virtual method `Task::execute()` will be overridden by the code of the software component;
- Each input of a task is realized by the class `Tasking::Input`;
- Each input object is associated with an object of the class `Tasking::Channel`;
- Each task may have multiple inputs and multiple outputs;
- A set of tasks, inputs and channels are framed in a scheduler entity, which is realized by the class `Tasking::Scheduler`;
- Each scheduler entity is provided with a scheduling policy;
- Each scheduler entity has threads to execute the assigned tasks according to the specified scheduling policy. The number of threads is specified by the software developer.
- Tasks can be activated also periodically by the means of the class `Tasking::Event`.

Although space software standards discourages virtual methods, the `execute()` method of tasks should be virtual to let the developer implement different tasks. A few other methods are intentionally virtual to add application code, e.g. synchronization of channel data.

To simplify setting up an object w.r.t. static memory management, we designed templates for the main classes.

A. Activation model

A task τ_i is activated and an instance of it will be queued when all inputs are activated (*and* semantic). *Or* semantic is also supported by providing the *final* flag. When the final flag is set for an input, the task will be activated regardless of other inputs.

The j-th input in_{ij} of task τ_i is activated when a predecessor task or other sources, e.g. the main thread, calls `Channel::push()` on the associated channel with in_{ij}. In the context of `Channel::push()`, the input in_{ij} will be set to active and if the final flag is set then τ_i will be activated, otherwise, the other inputs will be checked and τ_i will be activated only when all inputs associated to it are set to active.

Although we design our platform to be event-driven, time-triggered activation is supported by presenting the class `Tasking::Event`. Two time-triggered activation methods are supported: *periodic* and *relative time*. In the periodic method, the given time duration represents the distance between two successive events. Relative time method is used, for instance, when sensor data is needed. A task τ_{rc} sends a request command to the sensor then it sets the timer to a predefined time duration (relative time) and terminates. After the timeout occurrence, the following task τ_r reads the response sent by the sensor. Note that, this solution is similar to using *self-suspending* tasks [9]. Using relative time (in general using self-suspending tasks) requires to tightly bound the timeout. However, using channels connected to Interrupt Service Routines (ISR) of IO drivers (event-driven programming), in which τ_r is activated only when the sensor data is available, can improve the utilization. Figure 2 illustrates the relative time.

B. More features

1) Task group: The default call semantics among tasks that is supported in Tasking Framework is *asynchronous*, in which a task τ_i activates the successor tasks, then it can be executed again regardless of the status of the successor tasks. However,

in some applications, the graph of tasks or a subset of it has a *synchronous* call semantics such that τ_i activates the successor tasks and it will not be executed again till all tasks in the synchronous subset finish their execution. To support the synchronous call semantics, the class `Tasking::Group` is provided.

2) Task barrier: The number of activations at an input is declared at compile time. In situations, where the number of data elements is only known at run time, the activation cannot be adapted. This can be the case when, for example, a data source have states where no data is sent. The class `Tasking::Barrier` is a mean to control the activation of tasks with an unknown number of data packets.

By default the barrier can be instantiated with a minimum number of expected push operations on the barrier. After the minimum number of pushes happens, the barrier will activate all associated inputs, as long as data sources did not increase the number of expected push operations on the channel. If it is increased, more push operations are expected.

3) Unit test: We provide a special scheduler `SchedulerUnitTest` with step operation to support unit testing. Using Googletest (gtest) [10], we provide twelve classes to test the API.

Note that, the execution model has to be tested separately by the developer using other means, e.g. stress test.

III. Execution Model

Tasking Framework is a multithreading execution platform. The software developer should specify the number of threads, called *executors*. Therefore, there will be $n + 1$ threads: the main thread plus n executors. The implementation of the execution model is platform specific. We have three implementations of the execution model: the POSIX threading model (targeting Linux), C++11 threading and OUTPOST-core [2] (targeting RTEMS and FreeRTOS).

The execution model is represented by 4 classes:

- `Tasking::SchedulerExecutionModel`: Creating, scheduling, managing the executor threads and interfacing to the API.
- `Tasking::ClockExecutionModel`: Managing the time for time events. In embedded Linux, the clock is represented by a thread.
- `Tasking::Mutex`: An encapsulation of the mutex.
- `Tasking::Signaler`: An encapsulation of the conditional variables.

Tasking Framework schedules the ready task instances to the available executors according to the following scheduling policies: First-In-First-Out (FIFO), Last-In-First-Out (LIFO), and Static Priority Non-Preemptive (SPNP). The software developer can assign a priority to each task to be used by the SPNP queue.

An executor thread goes to sleep, i.e. waits on a conditional variable, after being created till it gets a signal from the clock thread (or a timer) in case of time-triggered tasks, or from other sources, e.g. the main thread. Figure 3 shows the life cycle of an executor thread.

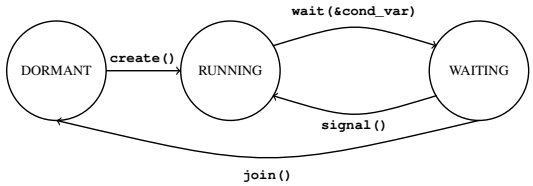

Fig. 3. Executor thread states

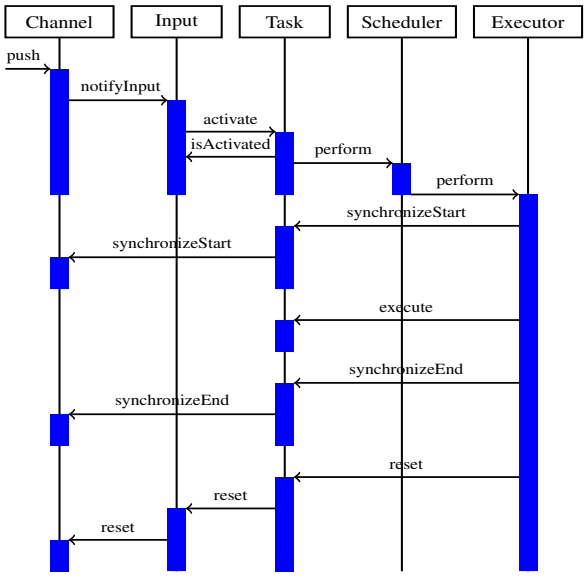

Fig. 4. The sequence of method calls in Tasking Framework to execute a task

The executor that executes the task τ_i activates the successor tasks and queue them in the ready queue, and it will signal a free executor, which is in *WAITING* state, if there is any. That is to say, Tasking Framework balances the load on the available executors. Even in case of one executor, the executor returns first from the `execute()` method of τ_i before checking the ready queue and executing the successor tasks of τ_i. The sequence of method calls that are performed by Tasking Framework to execute a task by an executor thread is shown in Figure 4. Because we have multiple threads that may try to access the data stored in the channel, a protection mechanism is implemented to synchronize the access to this shared data by different threads. The protection mechanism is implemented by the means of two virtual methods: `Channel::synchronizeStart` and `Channel::synchronizeEnd`.

In the implementation for Linux, the clock is implemented as a thread with the real-time clock provided by POSIX. The clock thread goes to sleep for a timeout equal to, e.g., the period of a periodic task. Then it signals a free executor if there is any, and it computes the next timeout.

A. Scheduling and priority handling

As has been mentioned, each instance of the `Tasking::Scheduler` is assigned a set of tasks,

inputs, channels, executors and a scheduling policy. With one instance for the application, the scheduling approach follows the global scheduling, i.e. all tasks can be assigned to any executor. However, it is possible to have multiple instances in one application. Considering the RTOS, we assign priorities to the executors (threads). Hence, we can handle priorities in groups (each group represents one `Tasking::Scheduler` instance).

IV. Tasking Modeling Language (TML)

We designed the API to be as usable as possible considering the high performance requirements of real-time onboard software systems. However, as the Tasking Framework is used in scientific missions with experts of different domains working on the system, the framework users might not be experts in implementing real-time software. To further improve applicability, we developed a model-driven tool environment that can be used to generate calls to the Tasking Framework API and its communication code to transfer data between different tasks. The tool is integrated into Virtual Satellite[1], a tool for model-based systems engineering. As Figure 6 shows, the TML development environment uses different types of description methods to model the software. Atomic data types are defined in tables, whereas data type classes and software components can be specified in textual languages. Because our focus is on data and event-driven communication, the connection of different components is modeled graphically. Each of the languages is specifically designed to describe software based on the Tasking Framework and, thus, further simplifies creating Tasking code.

A. Tasking-Specific Languages

Modeling languages specifically designed for a project or tool provide the benefit of introducing only few project-specific elements. The fewer elements in a language, the less effort is necessary to learn it. An early prototype of the modeling environment used the unified modeling language (UML) and the systems modeling language (SysML) to describe the Tasking-based software for the ATON project [11]. While the project clearly profited from modeling and its code generation, usage of the modeling tool required to understand UML, SysML and the Tasking Framework.

To improve modeling of software based on the Tasking Framework, we developed a tool suite including several domain-specific languages (DSLs) that contain all tasking relevant information. Figure 5 shows the editors of the different languages. The basic work flow is to define atomic data types first, then employ these to specify more complex data types, which are later generated as classes. As shown in the figure, atomic data types can be listed in a table. Additional attributes, such as the size of these types, allows running analyses about exchanged data and performance of components. After data types have been specified, it is possible to model software components. Components can be modeled either as plain classes or tasks and can have inputs, outputs and parameters.

[1]Virtual Satellite: a model-based tool for space system development; web page: https://www.dlr.de/sc/en/desktopdefault.aspx/tabid-5135/8645_read-8374/

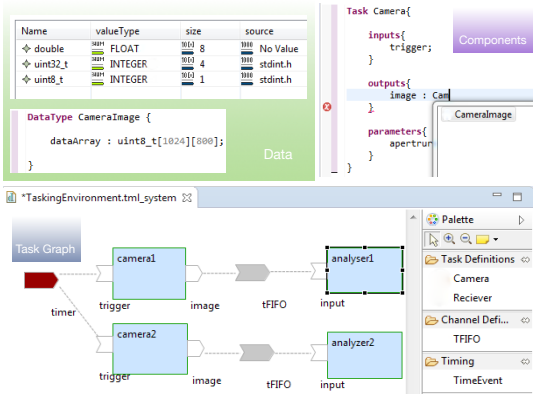

Fig. 5. Modeling tool environment for TML

After the description of data types and components, they can be instantiated in the main part of TML, the task graph. Instantiating components in this graphical description automatically adds inputs and outputs of the components. Connecting components validates data types and allows only compatible elements to be connected. Tasking-specific event parameters as well as timing and priorities of the components can be configured in this diagram.

Besides modeling of these main components of the Tasking Framework, it is possible to model custom communication channels, storage types, scheduling policies and units within the model.

B. Increased Development Productivity

The model-driven tool does not only simplifies the application of the Tasking Framework, it also increases the efficiency of developing software based on the framework. Projects with model-driven software development benefit from higher short-term productivity because users can generate new features from the model; long-term productivity increases because changing requirements can be handled by simply updating the model [12]. Thus, in context of the Tasking Framework, adding components to the diagram and connecting them generates their execution containers and communication code. This code does not have to be implemented manually. Furthermore, if project requirements change and the components have to be connected differently, reconnecting the elements in the diagram automatically updates the software's source code and documentation.

As Figure 6 shows, generation from the model not only generates source code but also build files and tests. SCons scripts or CMake files allow to build the generated code after generation immediately. This way, it is possible to start the development of a project from a running system and iteratively add new features.

C. Extensibility of the TML Model and Generated Code

To be applicable in as many projects as possible, the modeling environment is highly extensible. Besides defining

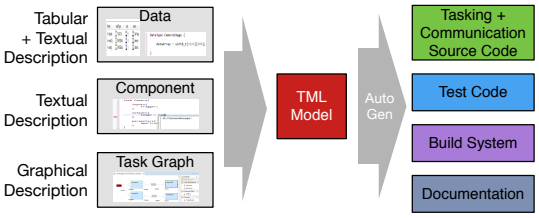

Fig. 6. Overview of the modeling environment and its generated atrifacts

custom data types and components, it is also possible to dynamically add new channel types with custom parameters. For such new channel types, the generator creates base classes with templates or constructor parameters depending on the parameters definition as static or dynamic. Instances of these custom types in the task graph are also generated as instances of the generated class and configured with parameter values from the model. Thus, even with a dynamic definition of new types and parameters within the model, the generated code remains executable.

In addition to the extensibility of the model language, the generated code can be customized by keeping code updates through regeneration possible. A combination of the decorator and generation gap pattern allows customizing the generated code by subclassing [13] [14]. The generated abstract class is regenerated, the concrete class is generated only once and then kept to not overwrite customizations.

V. Case Study

To demonstrate the benefits of the Tasking Framework in a simple real-time software, we recall our example in Figure 1. Figure 7 shows the architecture of the software as TML task graph diagram.

With the Tasking Framework, software components are implemented as tasks, data is stored in Channels and Events are used for periodic activation of the components. For the execution on the prototype flight computer, we assigned four threads to execute the software. As soon as the camera driver task pushes an image into the subsequent channel, the feature tracking task is notified and activated. The crater navigation task is configured to be activated only for every second image and, thus, runs with a reduced frequency. As the IMU driver does not have an external trigger, it is implemented as a thread that runs continuously and produces acceleration rate data with a frequency of 100Hz. Because the navigation filter has to update the output position with every IMU value, the final flag of its input data from the IMU is set. Therefore, if the input data from the crater navigation and feature tracking are available they are used, otherwise they are ignored.

To model this setup with TML for generation of the necessary tasking code, the first step is to define data types that can be used in the software. After a definition of the atomic types, such `double` and `uint8_t`, we can model the data types for `CameraImage`, `EstimatedPosition`, `AccelerationRate` and `NavigationState`. As next step, we have to model the actual components, which are generated as tasks. To create a model element for the `CameraDriver` task, we specify an input that does not have any data type and an output of type `CameraImage`. As last step of the element definition, we specify two different channel types: one LIFO channel with a parameter for its size and a channel with two buffers that switch every time data is added. In the task graph diagram we can then instantiate and connect all the previously defined elements. As we have two cameras, the camera driver task is instantiated twice. In the diagram we can then configure the timing and event parameters. As the crater navigation should run only every second image, we configure its input with a threshold of two. With the selected scheduling policy of priority-based, we can configure priorities for each task.

After we described the system in a TML model, we can generate its source code. All task definitions are generated with their in- and output interface; their instances in the diagram are created as objects in the software. Both cameras can be instances of the same camera driver task. For task definitions and custom channels, the generator creates base classes, which can be customized by subclassing.

VI. Related Work

Many platforms have been proposed for developing and testing embedded systems. Sadvandi, Corbier and Mevel presented in [15] a real-time interactive co-execution platform designed at Dassault Systèmes. The objective is to provide integration, co-execution and validation of heterogeneous models using model-based testing process, which comprises In-the-Loop testing, namely, Model-In-the-Loop (MIL), Software-In-the-Loop (SIL) and Hardware-In-the-Loop.

The Embedded Multicore Building Blocks (EMB^2) [16] is an open source C/C++ framework for the development of parallel applications. EMB^2 is developed by Siemens AG to efficiently exploit symmetric and asymmetric multicore processors. EMB^2 provides different scheduling strategies for both hard and soft real-time systems. Although Tasking Framework supports multithreading, it is not specifically dedicated for multicore systems.

OUTPOST is an open source mission and platform independent library developed in the German Aerospace Center (DLR) to design and implement reusable embedded software as early as possible and hence to be independent from the operating system and the underlying hardware. OUTPOST is originally called libCOBC, and it has been used in the Eu:CROPIS project [17], and in the ScOSA project [8]. Tasking Framework runs on the top of OUTPOST, and makes use of the services provided by it. One implementation of the execution model of the Tasking Framework is dedicated for OUTPOST as we have mentioned in Section III.

RODOS (Real-time On-board Dependable Operating System) [18], [19] is a real-time operating system developed at the German Aerospace Center (DLR) for network-centric core avionics [20]. Currently, RODOS is developed at University of Würzburg. The main goal of RODOS developers was to make it simple and dependable. The publisher-subscriber messaging

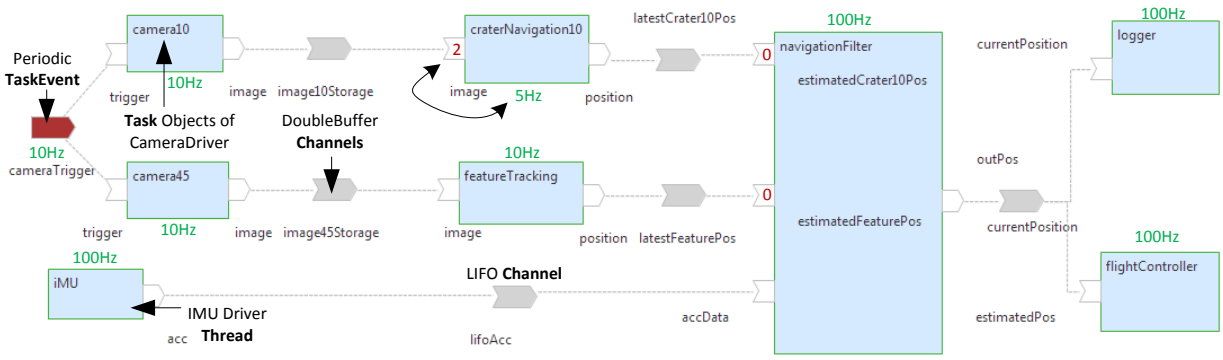

Fig. 7. TML system diagram of an application of the Tasking Framework in a real-time system for optical navigation in space.

pattern is considered in RODOS. In this pattern, publishers label messages according to predefined topics; one or more subscribers to a given topic receive all messages that are published under this topic. Unlike RODOS, a task in Tasking Framework pushes its output data into the associated channel and notifies the input of the next task/s with no call of the execute method of that task. However, Taking Frameworks is not an operating system.

VII. Conclusion

In this paper we presented our event-driven multithreading execution platform and software development library: Tasking Framework. It is dedicated to develop space applications which perform on-board data processing and sophisticated control algorithms, and have high computational demand. Tasking Framework has been used in already flying satellites, e.g., Eu:CROPIS.

Tasking Framework is neither a testing platform nor an operating system. It is a set of abstract classes with virtual methods to develop and execute data-driven on-board software systems on single-core as well as parallel architectures. It is compatible with the POSIX-based real-time operating systems, mainly RTEMS and FreeRTOS. Tasking Framework is supported with a model-driven tool environment (TML) that can be used to generate the API and its communication code.

Our plan is to make Tasking Framework open source. A bare-metal implementation is also on our agenda.

References

[1] G. Ortega and R. 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).

[2] German Aerospace Center (DLR), "Open modUlar sofTware PlatfOrm for SpacecrafT (OUTPOST)." https://github.com/DLR-RY/outpost-core. accessed 2019-04-14.

[3] K. Brie, W. Bärwald, F. Lura, S. Montenegro, D. Oertel, H. Studemund, and G. Schlotzhauer, "The BIRD mission is completed for launch with the PSLV-C3 in 2001," in *3th IAA Symposium on Small Satellites for Earth Observation* (R. Sandau, H.-P. Röser, and A. Valenzuela, eds.), pp. 323–326, Wissenschaft und Technik Verlag Berlin, April 2001.

[4] R. Olfati-Saber, "Distributed Kalman filtering for sensor networks," in *2007 46th IEEE Conference on Decision and Control*, pp. 5492–5498, Dec 2007.

[5] S. Theil, N. A. Ammann, F. Andert, T. Franz, H. Krüger, H. Lehner, M. Lingenauber, D. Lüdtke, B. Maass, C. Paproth, and J. Wohlfeil, "ATON (autonomous terrain-based optical navigation) for exploration missions: recent flight test results," *CEAS Space Journal*, March 2018.

[6] O. Maibaum and A. Heidecker, "Software evolution from TET-1 to Eu:CROPIS," in *10th International Symposium on Small Satellites for Earth Observation* (R. Sandau, H.-P. Röser, and A. Valenzuela, eds.), pp. 195–198, Wissenschaft & Technik Verlag, April 2015.

[7] B. Weps, D. Lüdtke, T. Franz, O. Maibaum, T. Wendrich, H. Müntinga, and A. Gerndt, "A model-driven software architecture for ultra-cold gas experiments in space," in *Proceedings of the 69th International Astronautical Congress*, pp. 1–10, 2018.

[8] C. J. Treudler, H. Benninghoff, K. Borchers, B. Brunner, J. Cremer, M. Dumke, T. Gärtner, K. J. Höflinger, D. Lüdtke, T. Peng, E.-A. Risse, K. Schwenk, M. Stelzer, M. Ulmer, S. Vellas, and K. Westerdorff, "ScOSA - scalable on-board computing for space avionics," in *IAC 2018*, October 2018.

[9] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen, "Many suspensions, many problems: a review of self-suspending tasks in real-time systems," *Real-Time Systems*, vol. 55, pp. 144–207, Jan 2019.

[10] Google, "Googletest - Google testing and mocking framework." https://github.com/google/googletest. accessed 2019-04-05.

[11] T. Franz, D. Lüdtke, O. Maibaum, and A. Gerndt, "Model-based software engineering for an optical navigation system for spacecraft," *CEAS Space Journal*, no. 0123456789, 2017.

[12] C. Atkinson and T. Kühne, "Model-driven development: A metamodeling foundation," *IEEE Computer Society*, 2003.

[13] E. Gamma, R. Helm, J. Ralph, and J. Vlissides, "Structural patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 196208, Addison-Wesley Professional, 1 ed., 1994.

[14] M. Fowler, "Generation gap," in *Domain-Specific Languages*, p. 571573, Addison-Wesley Signature, 2010.

[15] S. Sadvandi, F. Corbier, and E. Mevel, "Real time and interactive co-execution platform for the validation of embedded systems," in *9th European congress on embedded real-time software and systems*, 2018.

[16] Siemens AG, "Embedded Multicore Building Blocks ." https://embb.io/get_started.htm. accessed 2019-04-14.

[17] F. Dannemann and F. Greif, "Software platform of the DLR compact satellite series," in *Small Satellite Systems and Services Symposium*, 2014.

[18] S. Montenegro, "Network centric core avionics," in *2009 First International Conference on Advances in Satellite and Space Communications*, pp. 197–201, July 2009.

[19] S. Montenegro and F. Dannemann, "RODOS real time kernel design for dependability," in *Proceedings of DASIA 2009 Data Systems in Aerospace*, 2009.

[20] S. Montenegro, V. Petrovic, and G. Schoof, "Network centric systems for space applications," in *2010 Second International Conference on Advances in Satellite and Space Communications*, pp. 146–150, June 2010.