Design and Implementation of an Event-oriented Simulation in a Distributed System for Automated Testing of On-board Software

Workshop on Simulation and EGSE for Space Programmes (SESP)
28-30 March 2017

ESA-ESTEC, Noordwijk, The Netherlands

Annika Ofenloch[(1)], Fabian Greif[(2)]

[(1)]German Aerospace Center (DLR)
Robert-Hooke-Strasse 7, 28359 Bremen, Germany
Email: Annika.Ofenloch@dlr.de

[(2)]German Aerospace Center (DLR)
Robert-Hooke-Strasse 7, 28359 Bremen, Germany
Email: Fabian.Greif@dlr.de

ABSTRACT

Before launching spacecrafts, the on-board software (OBSW) and the communication of the on-board computer (OBC) with the subsystems must be tested to identify and correct possible errors in the software. Therefore, simulation environments are increasingly used in the development, verification and test phases. This allows the OBSW to be verified independently of the hardware or even before the avionics of the satellite is available [1].

The aim of this paper is to present the design and implementation of a simulator for automated testing of OBSW and the data exchange between the satellite components. A key requirement for the design is, to run the simulated avionics on different clients. This allows distributing the processor load and adding real hardware to the simulation at later development stages, making hardware-in-the-loop verification possible. The simulator can also save and load simulation states, which can be used to define mission scenarios and restore a simulation state at a later time.

INTRODUCTION

The simulation modelling platform (SMP) as described in the E-40-07 standard of ECSS does not explicitly support a distributed simulation. This results from the architecture of the simulation environment that provides simulation services (e.g. Logger, Time Keeper, Event Manager). Instances of these services are created once within the simulation environment and passed on to the other models of the system. An example for a software infrastructure that implements this standard is the ESOC Simulation Infrastructure for Satellites (SIMSAT). In order to use the SMP standard for a distributed simulation, a central network node is needed that transfers the packets based on the configuration to the corresponding subsystem. In other words, the simulation services will be sent to the switching node, which then takes care of the forwarding [3]. Due to its design the SMP environment also requires a mechanism for time synchronization to ensure that all subsystems on separate clients retrieve the identical time [2]. Against this background, a new approach is to be developed for a distributed simulation environment.

In this paper, the simulation is carried out with software models, which simulate the functional behavior of the system to be tested. The power control tasks of the Eu:CROPIS-satellite are taken as an use case, to test the OBSW and demonstrate the functionality of the simulator. Eu:CROPIS (Euglena Combined Regenerative Organic Food Production In Space) is a mission from the compact satellite program of DLR. The primary payload is conducting research in growing food under space conditions, that would enable scientists to produce food for use by humans on long duration missions, as well as on lunar and Mars habitats. This research includes an experiment on a closed life-support system that will be tested and validated on the Eu:CROPIS-satellite. It simulates the conditions found on the moon and Mars [4].

SIMULATION ENVIRONMENT DESIGN

As a starting point, an appropriate software architecture is presented. This includes the data exchange in the distributed system and the simulation process to illustrate the interactions between the subcomponents of the system. Furthermore, the detection of critical simulation cycles and three simulation modes are discussed.

Data exchange

As it can be seen in Fig. 1, the environment includes a simulation model, an event queue and custom models, representing individual subcomponents of the system being tested. In order to run each component on a separate client, communication between the subsystems takes place via TCP connections. In this work, the data should not be passed on to the relevant models via a switching node, as it is required for the simulation modelling platform of ECSS. Instead, the data exchange for the distributed system is realized using the publish-subscribe pattern of the network library ZeroMQ [5].

The simulation model starts the simulation and manages the simulation time t_i to distribute it further to other models via an event. This is implemented by ZeroMQ through a publisher (PUB) by making the events public. All other models can subscribe to these events through a subscriber (SUB). Thereby, the models receive only the events they are interested in. In addition, custom models can publish their own events defined within the models, which contain instructions when they should be executed.

Due to this design, the software models can be replaced by physical hardware during the development process. This allows the hardware to be tested together with simulated components as soon it is available. Thus, the simulation environment can be used for software-in-the-loop and hardware-in-the-loop verification.

Simulation process

Fig. 2 shows a possible simulation process, in which each sequence object either runs on another client or locally and communicates over TCP connections. Before the simulation starts, all models must be configured. At the same time, predefined events are loaded into an event queue and sorted via a scheduler by their time stamp. An event has an identifier, time stamp, data, and an indicator whether it is a periodic event.

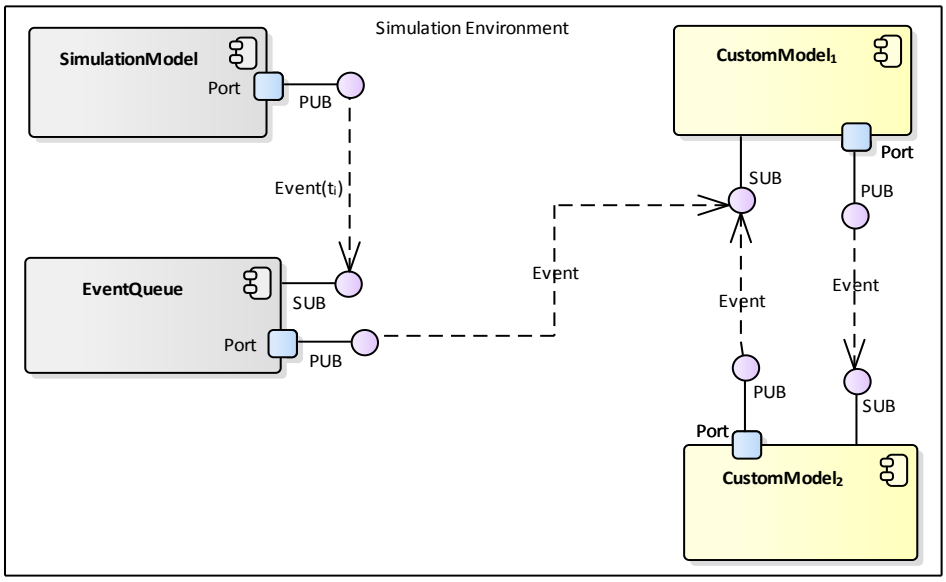

Fig. 1 Data exchange between the subcomponents of the system

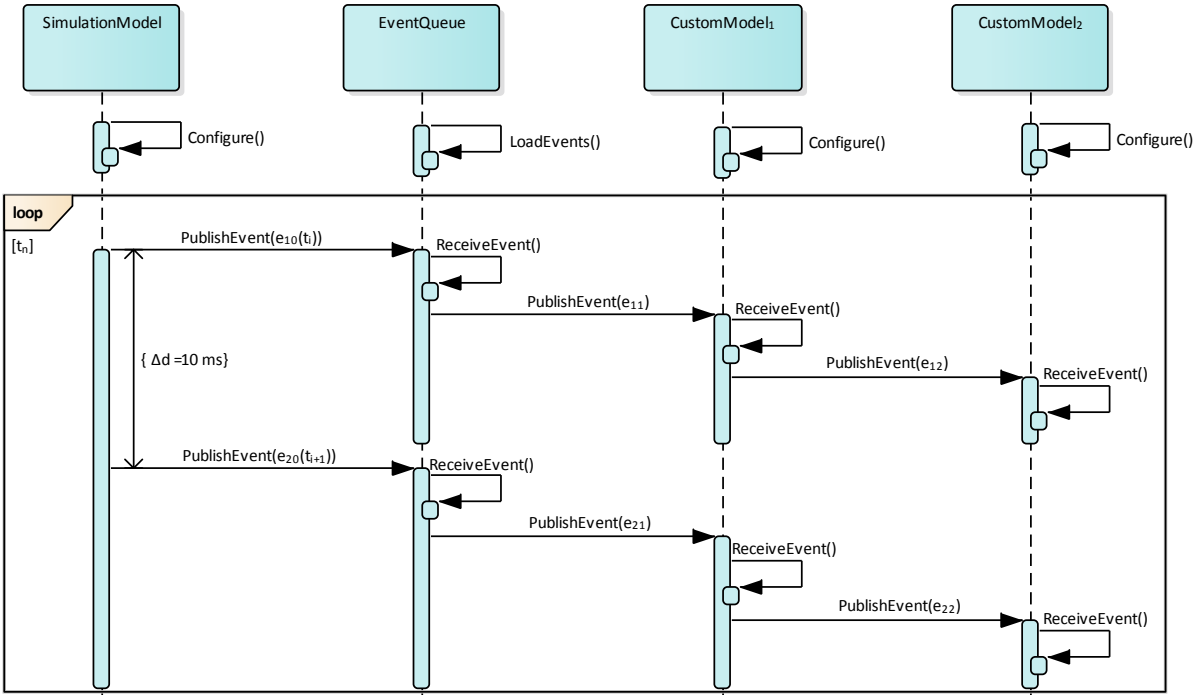

Fig. 2. Example of a simulation process

Once the models are configured, the simulation starts and publishes the current simulation time t_i via an event e_{01} to distribute it to the other models. Since the event queue has subscribed e_{01}, it receives the current simulation time, with which it can determine and publish the next event e_{11}. In the illustrated example, e_{11} is received by CustomModel_1. After e_{11} has been processed, CustomModel_1 publishes its own Event e_{12}, which has been previously defined and is triggered at certain states or actions. Subsequently, the last event e_{12} of the cycle is received and processed by CustomModel_2.

As soon as the defined cycle time Δd has elapsed, a new cycle starts at t_{i+1} by increasing the simulation time by Δt. A simulation cycle is performed until the simulation end time t_n has been reached.

Detection of critical simulation cycle

Delta cycles are performed within a simulation cycle, while no simulation time passes. During their execution events trigger other processes, which in turn execute new events.

Critical cycles may occur during the simulation process. This happens when a new simulation cycle starts at time t_{i+1} and the previous cycle at time t_i has not yet completed all delta cycles. Thus, the newly started simulation cycle may access parameters of a model which have not yet been updated in the previous time step. This leads to misinterpretations and error propagation during the simulation. Such a critical cycle, shown in Fig. 3, must be detected and intercepted.

Since the simulation model and the other models do not know when the last delta-cycle within a simulation cycle is completed, the critical sections must be identified by checking the time stamps of the received events. Each model must check whether the time stamp from the currently received event is smaller than the time stamp of the previously received events. If this is not the case, an event from the new cycle has already been executed and the simulation must be terminated. The cycle time can then be increased to provide more time to process the events within a simulation cycle and the simulation must then be restarted.

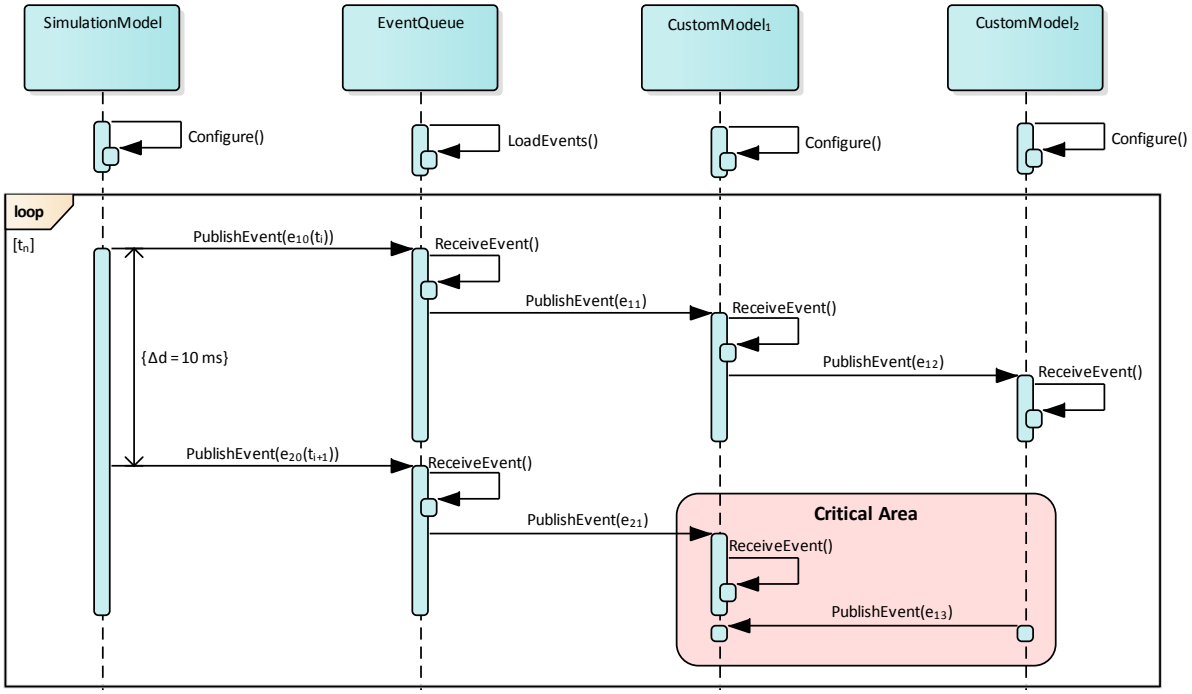

Fig. 3. Simulation process with a critical cycle

Simulation Modes

The simulation can be started in three different modes. This includes the execution in real time as well as faster and slower than real time, as it can be seen in Fig. 4. The simulation process shown on the left is slower than real time, the one in the center runs in real time and the simulation process shown on the right is faster than real time. All three modes have the same simulation time t_i and simulation time step Δt, but a different cycle time Δd. The smaller Δd is, the faster the simulation will be run and vice versa. If the simulation time step Δt is equal to the cycle time Δd, the simulation runs in real time. The cycle time can be calculated via $\Delta d = \Delta t / s$ by using the parameter s, which indicates the speed. The smaller s is, the larger Δd will be and vice versa. The simulation runs in real time, if s is equal to one.

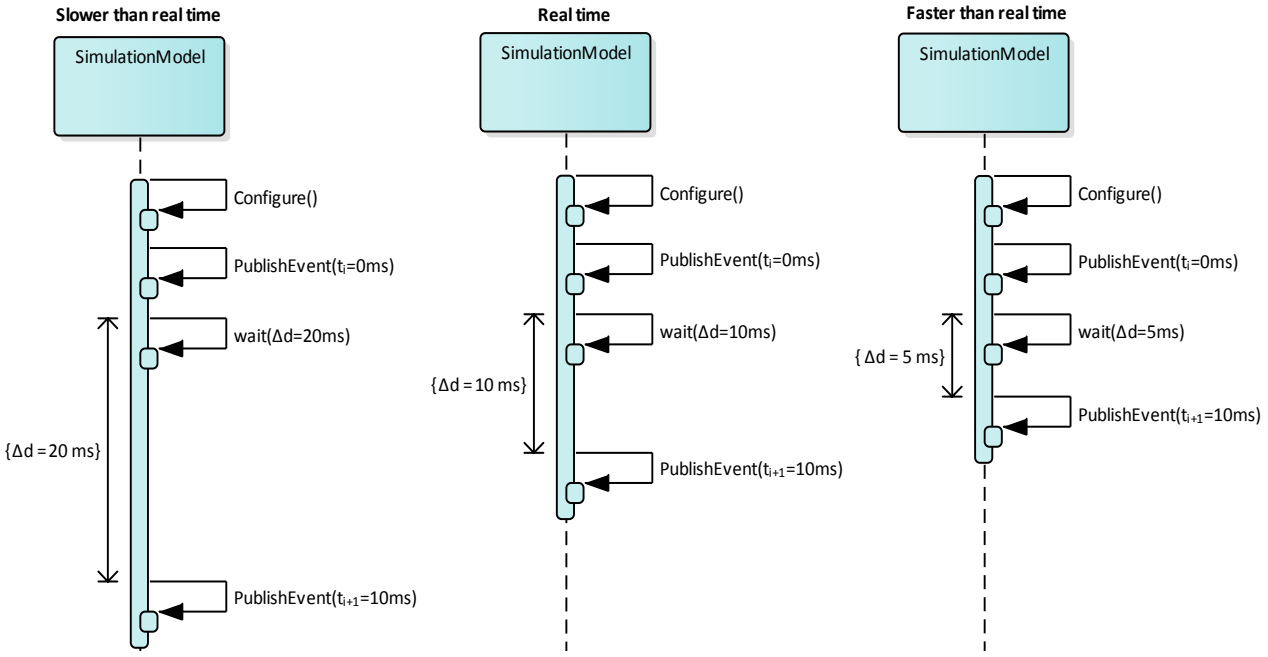

Fig. 4 Simulation modes: (left) Slower than real time; (center) real time; (right) faster than real time

Eu:CROPIS-satellite as an use case

The satellite bus contains the on-board computer, on which the flight-software is executed. It is connected with all subcomponents of the satellite and provides data handling services, receives commands and generates telemetry. As it can be seen in Fig. 5, the Power Conditioning and Distribution Unit (PCDU) is connected via a serial interface to the OBC. The PCDU performs power control tasks, which includes the acquisition of telemetry like current and voltage.

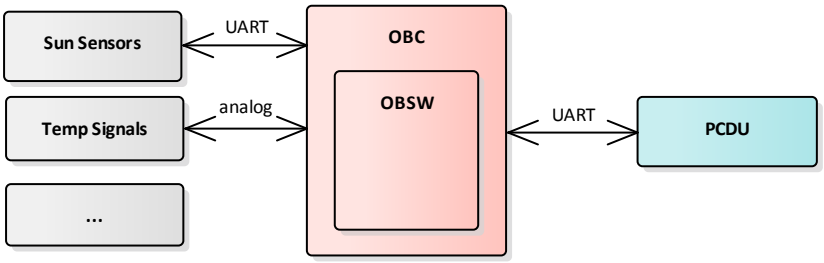

Fig. 5 Part of the On-board Computer block diagram

A possible implementation of the simulation environment can be found in Fig. 6, referring to the Eu:CROPIS-satellite. A flight software simulation is used and integrated into the simulation environment to test the OBSW. The OBSW uses a hardware abstraction layer (HAL) called “Device Driver Factory” to access the connected physical or simulated hardware. To integrate the simulation into the HAL, it requires a clock model which specifies the time since the OBC started. The OBC starts at the same time as the simulation. Thus, the clock model must get the current simulation time, which is converted by the OBSW to its internal representation and passed on to the other devices.
Furthermore, a serial model (PCDUSerialModel) is needed to provide an interface to the device model (PCDUModel), which performs the power control tasks. During operation, PCDU commands are generated and sent to the device model via the serial model. The device model includes a virtual PCDU that has the functionality to process the received command and generates the corresponding telemetry data. This data is sent back to and temporarily stored in the serial model. When the flight software simulation evaluates the PCDU telemetry data, the buffered data is read out and returned from the serial model. Additional events can be triggered through the event queue to modify the model states. This is used to analyze the behavior of the system in case of errors or changes.

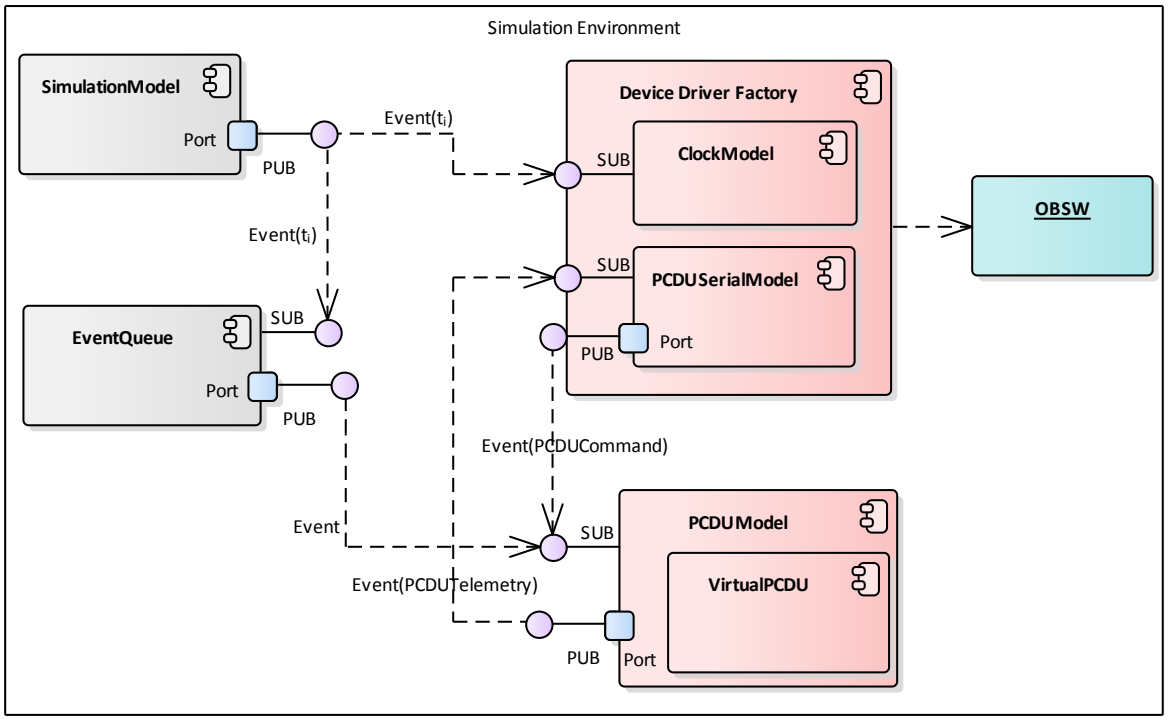

Fig. 6. Simulation environment for the Eu:CROPIS-satellite

CONCLUSION

This paper presents an event based simulation environment that can be used to test and validate various systems, for example spacecrafts such as the Eu:CROPIS-satellite. The subcomponents can be run on separate clients by communicating with each other over TCP connections. This distributes the processor load and makes it possible to replace software models with physical hardware. Currently the simulation is carried out and tested with software models, which simulates the functional behavior of the system. Moreover, the simulation can run at different speeds.

The difference to the SMP environment within a distributed system lies in the data exchange. In contrast to the use of SIMSAT, no central network node, which transfers the data to the corresponding model, is required in the presented simulation environment. Instead, the data is sent out by a publisher without knowing the subscribers. The models exclusively receive the data for which they have expressed their interest.

If physical hardware is added in later development stages, the internal clocks of the devices must be synchronized with the simulation time. For keeping the jitter low, the simulation time should be pre-computed within each subsystem. Thus, the simulation environment can be used for hardware-in-the-loop verification as well.

REFERENCES

[1] ESA Requirements and Standards Division, "ECSS-E-TM-10-21A Space engineering - System modelling and simulation," 16 April 2010.

[2] F. Cordero, J. Mendes, B. Kuppusamy, T. Dathe, M. Irvine and A. Williams, "A Cost-Effective Software Development and Validation Environment and Approach for LEON based Satellite & Payload Subsystems," *Proceedings of 5th International Conference on Recent Advances in Space Technologies - RAST2011,* pp. 511-516, 9.-11. June 2011.

[3] M. Pignède, J. Morales, P. Fritzen and J. Lewis, "Swarm Constellation Simulator," *SpaceOps 2010 Conference,* 25.-30. April 2010.

[4] F. Dannemann and F. Greif, "Software Platform of the DLR Compact Satellite Series," *Proceedings of 4S Symposium 2014. 4S Symposium,* 26.-30. Mai 2014.

[5] P. Hintjens, ZeroMQ: Messaging for Many Applications, O'Reilly Media, Inc., 2013.