A Flexible Distributed Simulation Environment for Cyber-Physical Systems Using ZeroMQ

Annika Ofenloch and Fabian Greif
Institute of Space Systems – German Aerospace Center (DLR), 28359 Bremen, Germany
Email: Annika.Ofenloch@dlr.de; Fabian.Greif@dlr.de

Abstract —Cyber-Physical Systems (CPS) are becoming increasingly complex. Their development and evaluation are carried out by several teams at different sites, while the time and budget is limited. Costly delays can occur, when the interplay of subsystems is to be tested and certain hardware components are not continuously present on site. Before CPS can be put into operation, they must be tested for functionality, reliability and safety. Possible errors must be detected and corrected at an early stage, both in software and hardware. Therefore, simulators are increasingly used in the development, verification and test phase. By replacing parts of the CPS with a simulated variant, hardware and software components can be developed in parallel at different locations by various organizations. The aim of this paper is to present a distributed event-based simulation environment for CPS that is reusable across various organizations and easily expandable. The simulation is carried out with software models, which simulate the functional behavior of the CPS to be tested. Simulation models or interface adapters for hardware components can be developed using defined software interfaces, regardless of the chosen platform or programming language. They can be integrated into the simulation environment with minimal effort and executed on distributed computer systems, while the communication takes place via ZeroMQ. The simulation environment is particularly suitable for systems that require low latency to guarantee real-time performance.

Index Terms—Distributed Simulation, ZeroMQ, simulation environment, cyber-physical systems, co-simulation

I. Introduction

For inaccessible, cost-intensive or fictitious systems, models are created which represent a physical, mathematical or logical representation of the system [1]. According to ISO/IEC/IEEE 15288, a system is a combination of interacting elements organized to achieve one or more specified goals [2].

In order to get quantitative information about the behavior of a complex *Cyber-Physical System* (CPS), it is simplified to an abstract model. Therefore, the CPS is divided into its subsystems. For each subsystem, a software model is created that implements the respective functional behavior. This is usually done in small teams distributed at different locations.

Manuscript received January 2, 2017; revised May 15, 2017.
This work was supported in part by the EU under Grant No. 644905.
Corresponding author email: Annika.Ofenloch@dlr.de.
doi:10.12720/jcm.13.6.333-337

Against this background, a simulation environment for CPS with a loosely-coupled infrastructure is presented called FRASER (Flexible DistRibuted Event-Based Simulation EnviRonment), in which models can be easily integrated or replaced with minimal user interaction. Software models can be used to replace parts of the CPS to test subsystems and their interplay independently of other hardware components. This is particularly helpful if individual subsystems are not constantly available on site. Once the physical hardware is available, it can replace the software model to perform *Hardware-In-The-Loop* (HIL) verifications.

II. Related Work

Simulation is a wide research field and different tools, standards and approaches for the simulation of complex systems exist. In this paper decentralized simulation is examined to execute modelled subsystems distributed on different computers.

An early work of a distributed simulation environment comes from the SIMNET (SIMulator NETworking) project for military training in 1983 [3]. SIMNET is the precursor to the *Distributed Interactive Simulation* (DIS) that was approved as a standard (IEEE 1278) in 1993 [4]. Many basic principles, which are defined in SIMNET and DIS, are still included in the *High Level Abstraction* (HLA) approach, developed by the U.S. Department of Defense and approved as a standard (IEEE 1516) in 1996 [5]. The latest standard of HLA (IEEE 1516-2010) was released in 2010 [6]. The data distribution and other operations in HLA are carried out by a distributed operating system called *Runtime Infrastructure* (RTI) that provides several services. All interactions among the subsystems (local or remote) flow through the RTI [7]. This leads to a drawback in the performance, since in a decentralized simulation the data must be sent twice over the network. In order to achieve a better performance, a simulation environment is presented where the communication takes place directly between the subsystems.

Further approaches for distributed simulations are to be found especially in the field of smart grids and in the automotive industry. One example is the *Functional Mockup Interface* (FMI) for co-simulation initiated by Daimler AG [8]. FMI is used primarily in the automotive industry, but also in other areas such as aerospace. The

FMI for co-simulation assumes to have a master-slave structure, in which all interactions are handled by the master. Otherwise, there is a direct dependency between the subsystems. The master-slave structure has the advantage that the modelled subsystems are completely decoupled from each other. However, the disadvantage is that an additional communication step to the master is always required. This leads to a loss of performance. In addition, the master must be changed if the system composition is modified. In comparison, a simulation environment without a master-slave structure is introduced, in which the subsystems are still loosely coupled.

An example from the aerospace sector is the *Simulation Modeling Platform* (SMP) as described in the E-40-07 standard of *European Cooperation for Space Standardization* (ECSS) [9]. This standard enables effective reuse of simulation models and applications across space projects, minimizing the cost for the development of simulators. However, the SMP standard 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 [10]. An example for a software infrastructure that implements this standard is the ESOC *Simulation Infrastructure for Satellites* (SIMSAT) [11]. 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 [12]. In addition, the switching node has to be changed if the model hierarchy or model links are to be changed. Therefore, the SMP standard for a distributed simulation is suitable for projects where the architecture is not constantly changing and is clearly defined in advance [13]. It is not well applicable for simulations where models need to be exchanged, added or removed during the design and verification processes. For this, a simulation environment is needed that is able to react on design changes and can flexible switch between software models and hardware components.

Against this background, a simulation environment is presented where the communication takes place directly between the subsystems to decrease the number of network hops during the simulation. This is particularly suitable for systems that require low latency to guarantee real-time performance. At the same time, the tested system can be quickly configured and its composition modified with minimal effort without recompiling the application.

III. SIMULATION ENVIRONMENT DESIGN

The simulation environment FRASER consists of a simulation model (M_{Sim}), configuration server (M_{Config}), an event queue (M_{Queue}) and user-defined models M_i, $i \in D$, while D is the name set of subsystems of the CPS being tested. The component diagram in Fig. 1 shows the data exchange within a possible implementation. In the illustrated example, model A (M_A) and B (M_B) represent user-defined models.

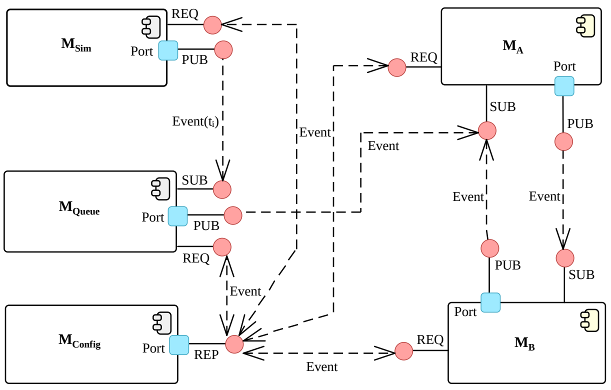

Fig. 1. Data exchange within a possible implementation of FRASER

A. *Data Exchange*

The simulation is event-based and controlled by the exchange of events [14]. An event has an identifier, a start time, data, and the indication whether it is periodic. An event queue contains events which are called at discrete points in time during the simulation. When a model receives an event, its states may be changed or new events may be triggered. The treatment routine for each event is defined within the models. Different event queues and initial states make it possible to simulate individual scenarios.

In addition, models can be executed on multiple computers. This distributes the processor load and models can be flexibly replaced by the physical hardware components using interface adapters, enabling HIL verifications. Interface adapters contain optionally hardware drivers and provide the interface to the simulation environment to enable a data exchange between hardware components and system level models. In order for models to run on distributed computer systems, the communication between the models takes place via TCP connections. This is performed by ZeroMQ, an open-source network library written in C under LGPL license by IMatix [15].

In traditional message queuing systems, such as ActiveMQ or RabbitMQ, there is a central message server (broker) with which all applications connect. There is no direct communication between the applications, but the messages are forwarded by the broker. As a result, each message must be sent twice over the network (from the sender to the broker and the broker to the receiver). In comparison, ZeroMQ is brokerless and does not need a central message server. The applications communicate directly with each other, so that messages can be transferred faster with fewer transmissions [16]-[18].

For the connection setup, required port numbers and public IP addresses are queried via a central network point (M_{Config}). To ensure that the models are not tightly coupled and still able to communicate with each other, the publisher subscriber design pattern as described in *Design Patterns: Elements Of Reusable Object-Oriented Software* [19] is used. ZeroMQ allows publishing events without making any assumptions about the recipients. Other models can subscribe to these events. Due to this design, new models and computers can be easily added to the simulation environment without great adoption efforts.

For example, a model for fault management that analyzes the state changes for each event execution can be easily added to the simulation environment. Therefore, all models publish their state via an event. The fault management model receives these states and compares them with previously defined ones. Detected anomalies are logged, making it easier to identify and localize errors in the CPS.

B. *Simulation Process*

The sequence diagram in Fig. 2 shows an example event-oriented simulation process. Before the simulation starts, all models must be configured and the connections must be established. At the same time, predefined events are loaded into an event queue and sorted via a scheduler by their time stamps.

Before M_{Sim} starts the simulation, it must be ensured that all models finished their configuration and no failures occurred. For this purpose, M_{Sim} begins periodically to publish synchronization messages, to which all models subscribe during their configuration phase. When they are ready for operation, they sent back a confirmation message. Once M_{Sim} has received all responses, it starts the simulation. However, if a model misses the time frame to response, the simulation is terminated due to a failed synchronization process.

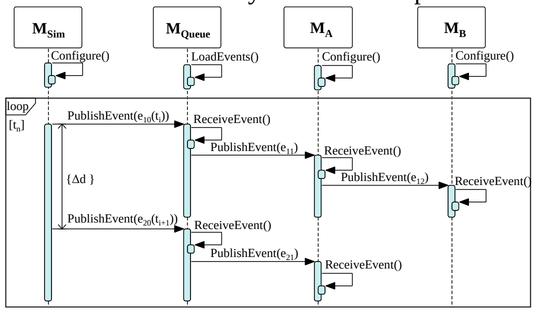

Fig. 2. Possible simulation process

If the synchronization process was successful, M_{Sim} starts to publish the current simulation time t_i via an event e_{01}, distributing it to the other models. Since M_{Queue} has subscribed to e_{01}, it receives the current simulation time, with which the next event e_{11} is determined and published. The execution of a subsequent event in the same simulation cycle is called delta cycle since no simulation time passes. In the illustrated example, e_{11} is received by M_A. After e_{11} has been processed, M_A publishes event e_{12}, which has been previously defined within the model and is triggered at certain states or actions. Next, the last event e_{12} of the cycle is received and processed by M_B.

As soon as the defined cycle time Δd, measured in wall clock time, has elapsed, a new simulation cycle starts at t_{i+1} by increasing the simulation time by Δt. One more time, the modified simulation time is communicated via an event. This triggers the next scheduled event within M_{Queue}. The simulation is performed until the end time t_n has been reached, where n indicates the number of simulation steps.

The simulation can be executed in real time as well as faster and slower 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 $\Delta d = \Delta t / s$, whereby s indicates the speed.

IV. Detection of Causality Errors and Livelocks

A. *Causality Errors*

Causality errors occur if events are executed in the wrong order [20]. This happens when a new simulation cycle starts at t_i and the previous cycle at t_{i-1} has not yet completed all delta cycles. Thus, the newly started 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.

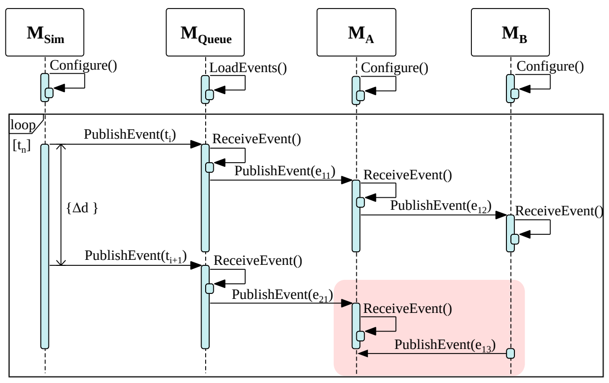

Fig. 3. Simulation process with a causality error

Since the simulation model and the other models do not know when the last delta cycle within a simulation cycle is completed, 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. Thereafter, the cycle time can be increased to provide more time to complete all delta cycles and the simulation must be restarted.

B. *Livelocks*

Livelocks occur when two or more processes are blocked. But unlike a deadlock, they do not remain in one state. Instead, they are constantly switching back and forth between several states [21].

As shown in Fig. 4, a livelock occurs when a cyclic dependency between two models in the data exchange is present and endlessly many delta cycles are executed. They will not necessarily lead to errors in other models, but indicate a general error in the defined system or models. If no events from the next simulation cycle are sent to the affected models, livelocks cannot be identified by checking the time stamps. Instead, it must be ensured that the simulation is terminated after a specific number of delta cycles. According to that, the event gets as additional information the current number of delta cycles.

Another approach is to stop the execution of delta cycles as soon as no relevant state changes occur. Therefore, the simulation will be carried out completely and not be terminated prematurely.

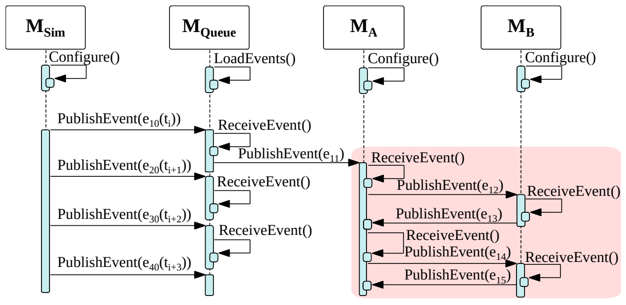

Fig. 4. Simulation process with a livelock

V. Satellite as an Use Case

As an example we are using a power control application as part of an *On-Board Computer* (OBC) of the Eu:CROPIS satellite from the *German Aerospace Center* (DLR) [22]. This application tightly integrates a safety-critical digital control system with its physical environment. The objective of the system is to manage the energy sources by controlling and distributing the power to the satellite's units. The OBC, on which the flight software (*On-Board Software* – OBSW) runs, is located on the satellite bus and connected to all subcomponents – providing data handling services, receiving commands and generating telemetry. In this example the *Power Conditioning and Distribution Unit* (PCDU) is connected to the OBC via a serial interface. The PCDU performs power control tasks, which includes the acquisition of telemetry like current and voltage. In order to test them, models for the OBC and PCDU are developed and integrated into the simulation environment.

An implementation of the simulation environment can be found in Fig. 5, referring to the Eu:CROPIS satellite [23]. 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 (M_{Clock}) 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.

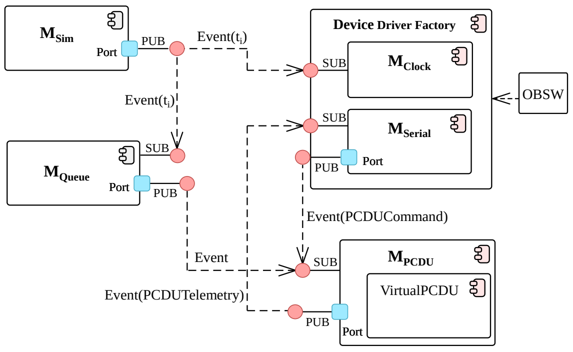

Fig. 5. Simulation environment for the Eu:CROPIS satellite

Furthermore, a serial model (M_{Serial}) is needed to provide an interface to the device model (M_{PCDU}), 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 state changes.

VI. Conclusion

The developed event-oriented simulation environment named FRASER allows a distributed simulation to test and validate complex CPS. FRASER is very extensible and has hardly any restrictions in the modeling of a CPS. Models can be executed on distributed computer systems, while the communication takes place via ZeroMQ with no message broker. Due to the publisher-subscriber pattern, the models are loosely coupled. This allows adding new software models to the simulation or replacing them by physical hardware components with minimal effort during the development process. The elegance of the simulation environment is its simplicity.

Causality errors are detected and intercepted when several events are executed from different simulation cycles within the models. In addition, two approaches have been presented with which endless cyclic delta cycles can be identified, without comparing the time stamps of the events.

The use of FRASER in the Eu:CROPIS mission of the DLR shows the power of the simulation environment. Especially the possibility to simulate error scenarios and to analyze the reactions of software and hardware components is of central importance.

In the future, the simulation environment will be used to simulate various fault management methods in order to test, modify and optimize new approaches. This is intended to enable complex critical systems to be operated autonomously.

ACKNOWLEDGMENT

This work was supported in part by the European Union (Horizon 2020 IMMORTAL project, grant no. 644905).

REFERENCES

[1] J. A. Sokolowski and C. M. Banks, *Modeling and Simulation Fundamentals - Theoretical Unterpinnings and Practical Domains*, Hoboken, New Jersey: Wiley and Sons, 2010.

[2] ISO/IEC/IEEE International Standard, Systems and Software Engineering - System Life Cycle Processes - ISO/IEC/IDEE 15288, 1st ed. Switzerland, 2015.

[3] D. C. Miller and J. A. Thorpe, “SIMNET: The advent of simulator networking,” *Proc. of the IEEE*, vol. 83, no. 8, pp. 1114–1123, August 1995.

[4] M. L. Loper, “Introduction to distributed interactive simulation,” in *Distributed Interactive Simulation Systems for Simulation and Training in the Aerospace Environment: A Critical Review*, vol. 10280. International Society for Optics and Photonics, 1995.

[5] R. M. Fujimoto, “Parallel and distributed simulation systems,” in *Proc. Winter Simulation Conference*, Arlington, VA, vol. 1, 2001, pp. 147-157.

[6] R. M. Fujimoto, “Parallel and distributed simulation,” in *Proc. Winter Simulation Conference*, Huntington Beach, CA, 2015, pp. 45-59.

[7] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly, “The department of defense high level architecture,” in *Proc. 29th Conference on Winter Simulation*, Washington, DC, USA, 1997, pp. 142-149.

[8] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, *et al*., “The functional mockup interface for tool independent exchange of simulation models,” in *Proc. 8th International Modelica Conference*, Dresden, Germany, no. 63, pp. 105-114. Linkoeping University Electronic Press, Mar. 2011.

[9] ESA Requirements and Standards Division, “Space engineering - simulation modelling platform: Principles and requirements,” in *ECSS-E-TM-40-07*, vol. 1, Jan. 2011.

[10] ESA Requirements and Standards Division, “Space Engineering - System Modelling and Simulation,” in *ECSS-E-TM-10-21A*, April 2010.

[11] J. Whitty, “Real time distributed simulations using SIMSAT 4.3,” in *Proc. Simulation and EGSE facilities for Space Programmes*, September 2010.

[12] 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,” in *Proc. 5th International Conference on Recent Advances in Space Technologie - RAST2011*, Istanbul, 2011, pp. 511-516.

[13] European Space Agency, *SMP 2.0 Handbook Issue 1 Revision 2*, EGOS-SIM-GEN-TN-0099, October 2005.

[14] A. M. Law, “Simulation modeling and analysis,” in *Industrial Engineering and Management Science*, McGraw-Hill Education, 2015.

[15] A. Dworak, M. Sobczak, F. Ehm, W. Sliwinski, and P. Charrue, “Middleware trends and market leaders 2011,” in *Proc. 13th International Conference on Accelerator and Large Experimental Physics Control Systems*, France, pp. 1334, October 2011.

[16] F. Akgul, “ZeroMQ,” Packt Publishing Ltd., 2013.

[17] L. Magnoni, “Modern messaging for distributed sytems,” *Journal of Physics: Conference Series*, vol. 608, no. 1, May 2015, pp. 012038.

[18] S. Celar, E. Mudnic and Z. Seremet, “State-of-the-Art of messaging for distributed computing systems,” in *Proc. 27th International DAAAM Symposium*, Jan. 2016, pp. 0298-0307.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Boston, MA: Pearson Education, 1994.

[20] R. M. Fujimoto, “Parallel discrete event simulation,” *Communication of the ACM*, vol. 33, no. 10, pp. 30-53, October 1990.

[21] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed., New York: Springer Science & Business Media, 2009.

[22] F. Dannemann and F. Greif, “Software Platform of the DLR Compact Satellite Series,” in *Proc. of the 2004 4S Symposium*, Mai 2014.

[23] A. Ofenloch and F. Greif, “Design and Implementation of a Discrete Event-oriented Simulation in a Distributed System for Automated Testing of On-board Software,” in *Proc. Workshop on Simulation and EGSE for Space Programs*, Netherlands, March 2017.

Mrs. Annika Ofenloch is a scientist in the German Aerospace Center (DLR) Institute of Space Systems. She pursued a professional education through a dual study program at DLR and has been engaged at DLR for more than 6 years. She received the B.S. degree from the Baden-Wuerttemberg Cooperative State University Mannheim, in 2014 and the M.S. degree from the University of Bremen, in 2017, both in Computer Science. She has a strong background in practical software design, development and testing for embedded systems.

Mr. Fabian Greif is the head of the Avionics Software Group in the German Aerospace Center (DLR) Institute of Space Systems. He is an expert for embedded system design and operating systems and is well experienced in electrical and software engineering. He holds a Dipl.-Ing. in Computer Engineering from the RWTH Aachen University. During his work at the DLR he has lead the software development for various Space Projects like the Eu:CROPIS satellite.