Development and Testing of Fast and Portable Data-Handling Models in a Synchronous Small Satellite Simulator

Kai Leidig*

University of Stuttgart Institute of Space Systems, 70569 Stuttgart, Germany

Leidig@IRS.Uni-Stuttgart.de

Bharadwaj Chintalapati† and Jens Eickhoff‡

Airbus Defense and Space GmbH, 88039 Friedrichshafen, Germany

Bharadwaj.Chintalapati@Airbus.com

On the basis of the small satellite *Flying Laptop* and its derived platform *FLP2*, this paper introduces a generic methodology of simulating a satellite on-board computer within an on-board data-handling network. The concept has been developed in order to take on some of the challenges towards a reference testing facility for small satellites. Although the development of a reference testing facility has been the initial motivation, this work does not intend to provide a full concept of such an architecture in any kind, but detail solutions to specific problems down that road.

The first challenge is to bind the controller-in-the-loop as close as possible to the satellite system simulator without unnecessary interfaces in between. This means that an appropriate simulator needs to provide all required interfaces, as it has to simulate all data-handling devices that are connected to the on-board computer in some way. Unfortunately, the working principle of a state-of-the-art satellite system simulator would causes latencies that rule out a proper simulation of an on-board data-handling, which is why a faster solution must be found here.

The second challenge is the derivation of a generic modeling approach for data-handling devices, which supports portability across different simulation projects. A solution here is restricted though by the variety of hardware architectures and protocols especially on transport and application layer, which is why this work sticks to the application of *SpaceWire* and the *Remote Memory Access Protocol*.

Verification of developed models is the third and last challenge met in the scope of this paper. Models of a satellite system simulator are implemented mostly in an object-oriented fashion. Testing object-oriented software does not work straightforward as a matter of principle. It is even more comprehensive, when the respective models are multi-threaded. Therefore, a profound approach of testing these data-handling models must be introduced.

*Research Assistant, Small Satellites Program, Simulation and Software Verification

†Satellite Systems Engineer

‡Honorary Professor of Satellite System Technology and Operations

NOMENCLATURE

CCSDS *Consultative Committee for Space Data Systems*
CDH command and data handling subsystem
CPU central processing unit
FLP *Flying Laptop* (Platform)
FLP2 *Flexible LEO Platform*
IF interface
IOB IO-board
IRS *Institute of Space Systems*
LEO low-earth orbit
OBC on-board computer
PCDU power control and distribution unit
PUS packet utilization standard
RMAP remote memory access protocol
SpW *SpaceWire*
STB system test bench
TC telecommand
TCP transmission control protocol
TM telemetry
TTC telemetry tracking and command subsystem
UDP user datagram protocol

I. INTRODUCTION

The work presented in this paper has been done in the scope of the development of *Flexible LEO Platform* (*FLP2*) [1] by *Airbus DS* in Friedrichshafen, Germany. *FLP2* has been derived from *Flying Laptop* [2] the first satellite of *University of Stuttgart Institute of Space Systems*, which was launched on July 14th, 2017 from Baikonur, Kazakhstan.

A. *Flying Laptop*

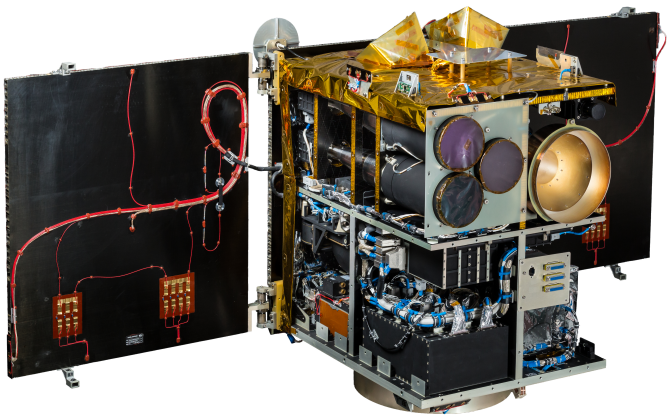

Figure 1. Small Satellite *Flying Laptop* - ©*IRS*

Flying Laptop is the first satellite of the small satellite group of *Institute of Space Systems*. It is an approximately 120 kg, three-axis stabilized satellite and the first application of the *FLP* small satellite platform. Over the past years it has been developed, integrated and tested by PhD and undergraduate students at the institute and was therefore research and training object on many occasions. With the satellite being launched into a 600 km sun-synchronous orbit, the purpose of the mission is manifold. One

major task has been since the in-orbit verification of the developed bus components, most prominently the command and data-handling subsystem (CDH) [3].

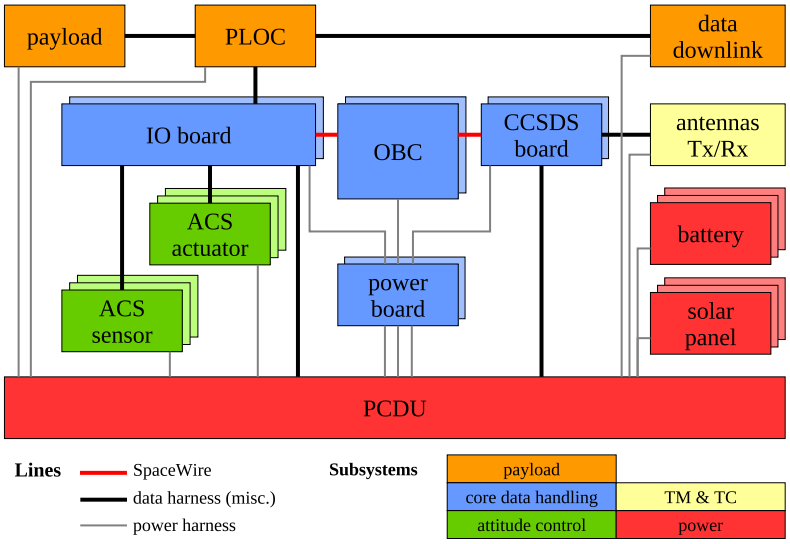

Figure 2. ***Flying Laptop*** **simplified harness scheme [3], ©*IRS***

Figure 2 depicts the simplified harness scheme of *Flying Laptop*. The entire bus is controlled by an on-board computer (OBC) with two cold redundant single-core LEON processor boards. These processor boards, two remote interface units called the IO-boards, and two CCSDS-boards[a] for TM/TC communication form the command and data-handling subsystem of *Flying Laptop*. Any logic device of the bus is connected to that subsystem. The connection is realized by means of the IO-board providing several interfaces for each specific device.

The energy household of FLP is managed by means of a dedicated logic device called the power control and distribution unit (PCDU). This device controls the battery charging by the solar panels and provides the power supply for all the electrical devices on board. *Flying Laptop* is powered by three battery strings with a total maximum capacity of 35 Ah and three solar panels generating a bus voltage of approximately 25 V.

For payload operations *Flying Laptop* uses a second computer called the PLOC, which can be controlled by the CDH via the IO-board. Basically, the PLOC is in charge of the entire payload subsystem, which means that it provides access to the payload components and manages the down-link of payload data via a dedicated down-link system called DDS.

B. The *Flexible LEO Platform*

Like *Flying Laptop* the new *Flexible LEO Platform* is fully three-axis stabilized. It is designed for a minimum lifetime of five years in a 800 km low-earth orbit. The platform shall suit satellites with a mass range between 50 and 200 kg, featuring configurable design options such as cube, hut, multi-box, etc. In extension of the *FLP* attitude control system, *FLP2* provides a propulsion system that enables change of altitude as well as station-keeping in scope of a constellation operation for instance. Considered are various propulsion concepts like cold-gas, chemical or electric systems.

Compared to *FLP* the major advance of *FLP2* is the command and data-handling subsystem. The platform features a new radiation hard LEON3 dual-core processor-board which controls the platform subsystems[b] on one core and performs the payload management on the second one. The payload management core is therefore independent and can also be used to process the payload data if needed. This allows for a reduction in the number of computing boards on the satellite and correspondingly reduces the number

[a] *Consultative Committee for Space Data Systems* - standard for satellite communication
[b] attitude control, thermal control, etc.

of redundant boards necessary. As *Flying Laptop*, the platform is fully CCSDS/PUS [4] compliant, which enables the spacecraft to communicate with the majority of commercial ground stations in the world.

FLP2 is based on a platform-wide *SpaceWire* [5] network, which complements the serial connectivity provided by the IO-board already used within *FLP*. Furthermore the platform shall support Ethernet devices by providing a respective *SpaceWire*-Ethernet bridge and a payload data downlink based on the DVB-R2/S2 protocol.

C. The *FLP2* Test Bench

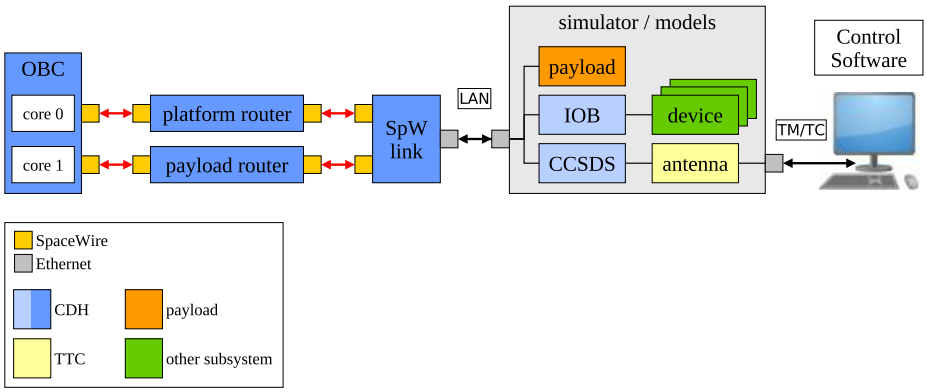

Figure 3. *FLP2* system test bench

For *FLP2*, a test bench was built at *Airbus* in Friedrichshafen, Germany which has been derived from the original *FLP* test bench located at *Institute of Space Systems* in Stuttgart, Germany. As *FLP* moved to its second generation, the test bench at Airbus began to be expanded and differ more and more from the original. A simple schematic of the *FLP2* test bench is shown in figure 3. The *FLP2* test bench is a hybrid STB which to date incorporates the real OBC, the PCDU (not depicted), *SpaceWire*-routers, the satellite simulator and the mission control software *CCS5*[c].

Normally, telecommands are encoded prior to the transmission to the satellite and decoded later by means of the CCSDS-board. In this setup the CCSDS-boards are simulated as well as most of the remaining satellite equipment. After TC reception, the simulated CCSDS-boards forward the commands to the CDH equipment that is not simulated. The simulation is realized by means of an *Airbus* in-house tool called *SimTG* which is also widely used in the production of several other satellite missions at the company. Section II will introduce the simulation of the platform CDH in more detail.

The STB is used for controller-in-the-loop tests as well as, in a first extension stage, for hardware/software compatibility tests. This means that in such a test setup, for the first time in the development process, the on-board software is loaded onto the real OBC. Hence, elementary hardware functions like booting, switching between redundant hardware instances, as well as TM/TC data-handling can be tested. This STB shall benefit from the findings introduced in this paper, since the described approach is supposed to be implemented within any data-handling model of the simulator depicted in figure 3.

D. Envisaged Application

Possible applications of *FLP* and *FLP2* derivatives are manifold. One prominent application envisaged is the usage of the platform(s) for constellation satellites. Focusing on that goal, *Institute of Space Systems* is currently developing a test bench for constellation operation research. In such a test bench not just one, but a number of satellites are simulated and commanded from a sophisticated mission control environment using *CCS5*.

Simulating a larger number of satellites, instead of just one, demands for a capable data-handling simulation even more. This is because in such a setup all of the satellites are completely simulated, meaning

[c]Central Checkout System

that all of the operated on-board computers have to be simulated too. A lack of a respective CDH modeling approach would rule out the development of such a test bench for constellation operation.

II. GENERIC SIMULATION OF CDH MODELS

Over the past year the authors have evaluated several configurations for the simulation of the *FLP* command and data-handling subsystem. One solution is depicted in figure 4.

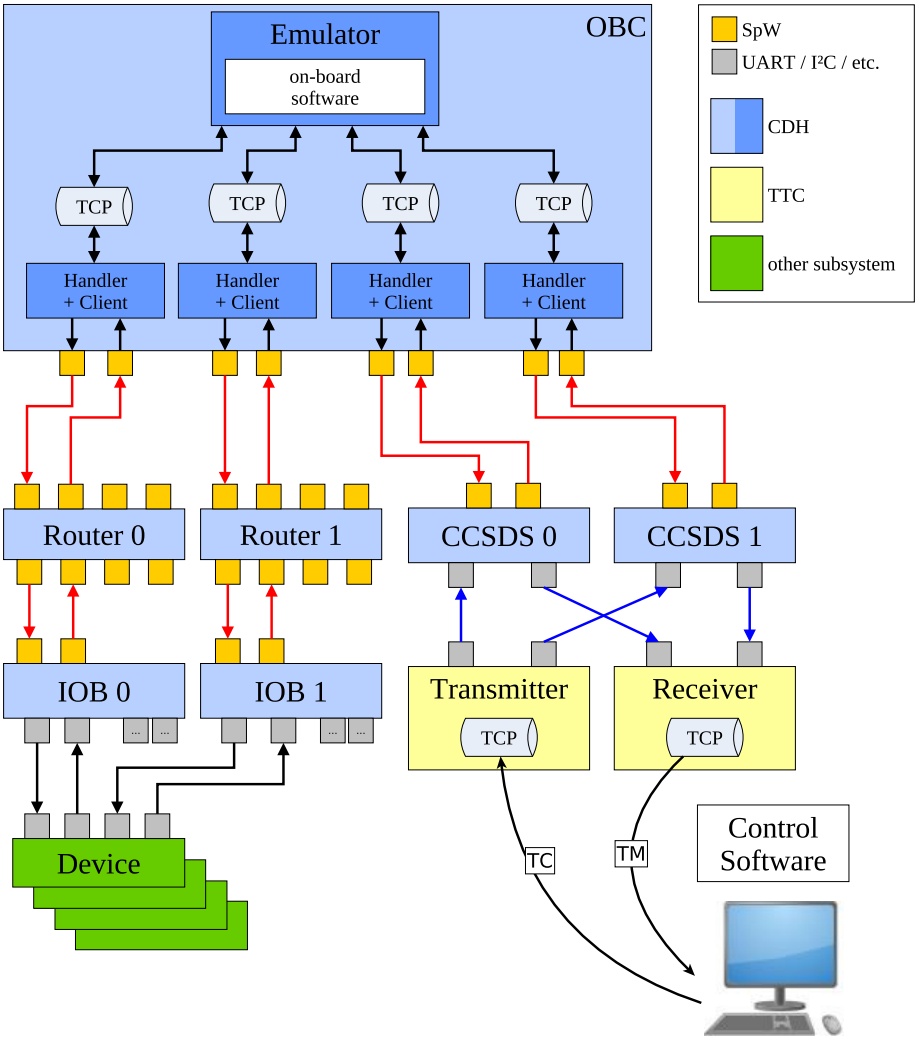

Figure 4. Example solution for simulating an *FLP*-like CDH, ©*IRS*

Core of the CDH is the on-board computer (OBC). Depending on the setup of the test bench and the application of the simulator the on-board computer is either simulated or the on-board software is hosted on a real development board. In the depicted configuration the on-board computer is simulated by means of a real-time hardware emulator. Wrapped around the emulation is the simulation model of the on-board computer which also performs the time-synchronization between the simulator and the emulation. It further provides means of accessing the emulation as well as forwarding the transmitted packets between the on-board software and the connected models. The major focus of this paper lies on a detail description of the internal data-handling by such or a similar model. It will be introduced in detail by the following subsections.

Attached to the on-board computer model on the one hand are the two CCSDS-boards, buffering and converting TM/TC packets for ground communication. On-board computer model and CCSDS-board models are linked by simulated *SpaceWire* lines, spanned between the respective interfaces of the CDH models. The

transferred *SpaceWire* packets are of the RMAP [6] standard. TM/TC packets composed and unpacked by the CCSDS-board model are transferred via separate transmitter and receiver models. These models use TCP sockets and an Ethernet connection for the communication with the mission control software.

The connection between the on-board computer model and the IO-board model on the other hand is realized in the same manner. The IO-board is the remote interface unit of *FLP* and its derivatives. It establishes the data connection to any logic device that does not speak *SpaceWire*. Since *FLP2* intends to promote *SpaceWire* for an extensive use within the entire bus, a router can be applied between the OBC and the IO-board. This allows attachment of further *SpaceWire* devices such as respective payloads (not depicted in figure 4).

As the OBC model, all the other CDH models (IO-board, CCSDS-board, router) apply the same methodology for internal data-handling. The models only vary in the way they process and buffer the data internally. The specific data processing procedures are not discussed by this paper though. This section focuses instead on the general description of the approach developed to simulate devices of a command and data-handling subsystem. Because the main goal of this approach was the adaptability to a various set of CDH devices, it is kept very simple and generic. Thus it does not matter whether the simulated device is some sort of CPU, a router or an interface unit etc.

A. Continuous (Layered) Approach

Usually, satellite systems are modeled and simulated in a synchronous fashion. Each model of a synchronous simulation has a step method that is called periodically after a discrete time step. This methodology is beneficial in terms of numerical simulation or in order to implement the state machine of a certain device.

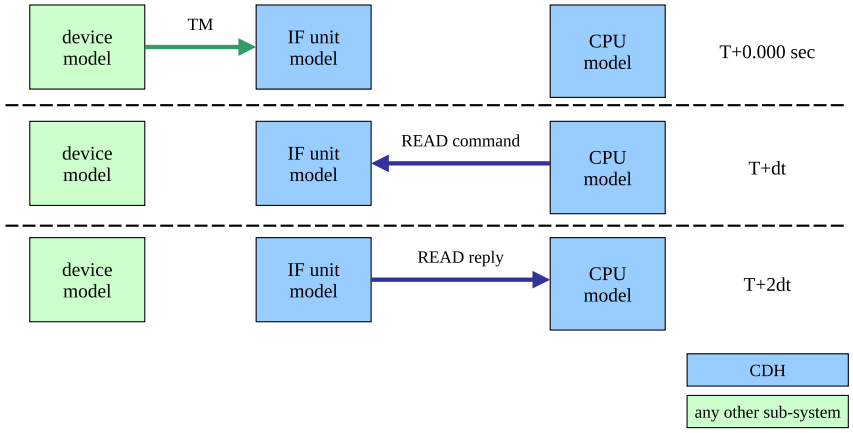

Figure 5. Timing issues due to a synchronous modeling approach

In order to model the internal data handling of a satellite, this approach is rather disadvantageous though. This is due to the fact that synchronous modeling causes latencies in the data handling process that rule out the full simulation of CDH devices. The example depicted by figure 5 shows the reason for such a delay quite clearly. In that case a device generates TM and sends it to the CDH at a certain time $T + 0.000\,\text{sec}$ for further processing by the on-board computer. Some sort of interface unit of the CDH needs to buffer that data, which requires another time step dt until the device is able to handle the read command from the on-board computer. The execution and the forwarding of the read reply then takes a further time step.

Synchronous models are scheduled usually with a period in the order of milliseconds. Standards such as *SpaceWire* however require responses within time spans that are about 10 to 100 times shorter [5]. Hence, a synchronous approach is way to slow to match the expected response times, even for a simple setup as depicted by figure 5. Response times can actually be much longer, if not a simple setup but a larger network is simulated.

The most obvious approach to circumvent this issue could simply be the increase of the scheduling frequency of the respective step methods in the CDH models. This however would be rather inefficient, because most of the model functionality implemented in the step function doesn't require execution at high frequencies. High frequent step function calls also might be limited by the scheduling capabilities of

the respective simulation framework. Therefore a more efficient methodology must be found to match the timing requirements.

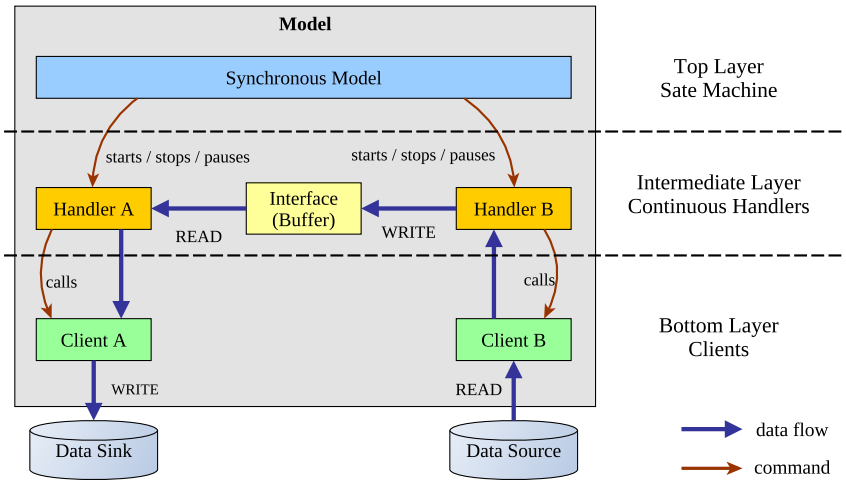

Figure 6. Layers of the simulated CDH model, ©*IRS*

A more convenient approach that covers these timing issues is shown in figure 6. The figure depicts the structure of a device model from a data handling point of view. At the top there is the state-of-the-art synchronous model, implementing the physical behavior and the state machine of the device. Underneath there is an intermediate layer of so called continuous handlers. These handlers are realized by means of separate, independent threads. These threads are instantiated and controlled from the synchronous simulation model that lies on top of them. Main tasks of the handlers are the continuous data reception from a data source, the identification of the target interface or buffer, and the writing of data on that target. Like there are handlers for data retrieval, there are also handlers for accessing the buffers and sending the data to a target data sink.

Data sources and data sinks can be accessed upon command via configurable clients, which constitute the bottom layer of this approach.

The separate layers are introduced in more detail by the following sections.

B. Synchronous Model - Top Level

On top of everything lies the actual model of the simulated device. Like all the other models it is called periodically in the scope of a synchronous simulation. As a class it defines all the attributes of the simulated device, which are ...

- physical parameters (nominal power consumption, operational temperature range, ...)
- model inputs/outputs (temperature, dissipated power, ...)
- model interfaces (data lines, power lines, ...)
- methods
- all parameters to configure the lower layers

In the scope of the initialization process that stands at the beginning of each simulation, the synchronous model configures its continuous handlers. Because the clients are attributes of the handlers the model needs to provide all configuration parameters for these clients too. The same applies to the configuration of the buffers/interfaces which the handlers need to communicate with. Depending on the kind of the simulated model the number of handlers, clients and internal interfaces can vary.

After the model initialization the major task of the synchronous model is to operate the handlers. For that purpose a state machine needs to be implemented within the model that can start, pause, or stop the

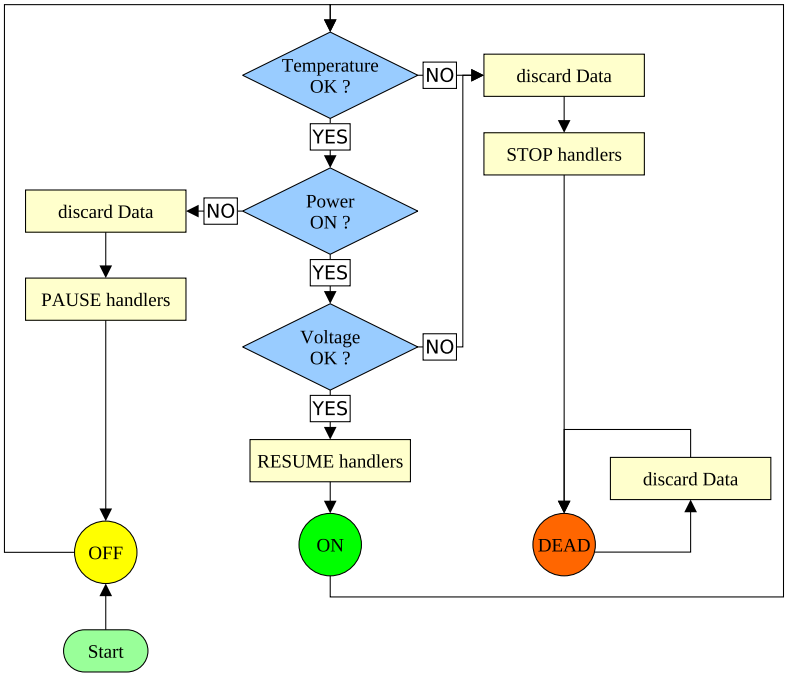

Figure 7. Physical state machine of a CDH model, ©*IRS*

handler threads. An example state machine is depicted in figure 7. At the beginning the device is in the *OFF* state, which means that the handler thread was created, but has been paused. If an initial temperature check is successful and the supply voltage is within the operational range the device is switched *ON* and the handler thread can be resumed. When no voltage is applied to the model power interface, the handler remains paused and all data from the model data source is discarded. In case too much voltage is applied to the model power interface or the temperature is out of limits, the device is considered *DEAD*. In that case handlers are permanently stopped and data will always be discarded. As figure 7 shows, the model cannot leave the *DEAD* state.

C. Continuous Handler - Intermediate Level

The working principle of the continuous handler is quite simple. Continuous handlers for a specific device are derived from a generic *C++* class that declares the virtual method `handleData()`. Handlers inheriting from this generic class must define `handleData()` according to the device functionality. An example implementation is depicted by figure 8. When the thread is running the method is called continuously.

At the beginning a packet is received. Depending on the direction of the data flow, the packet can either originate from an external data source or an internal interface (see figure 6). If the packet comes from an external data source, it is received by means of a client (subsection E). Otherwise the packet is received from an interface which can be read out directly (subsection D).

Packets that originate from an external source need to be checked. Respective definitions of `handleData()` implement a checking of the received packet, the checking is conducted by the client though. The client performs the checking since it implements the used protocol anyway. *FLP* and its derivatives use the remote memory access protocol (RMAP) [6] for internal CDH communication.

Successfully checked packets need to be written to the correct target. Again, targets can be an external data sink or an internal interface or buffer. If more than one target is available, the correct one needs to be identified by means of the information provided by the transferred packet. According to external data sources, external data sinks can be accessed with a dedicated client. Internal interfaces can be written directly.

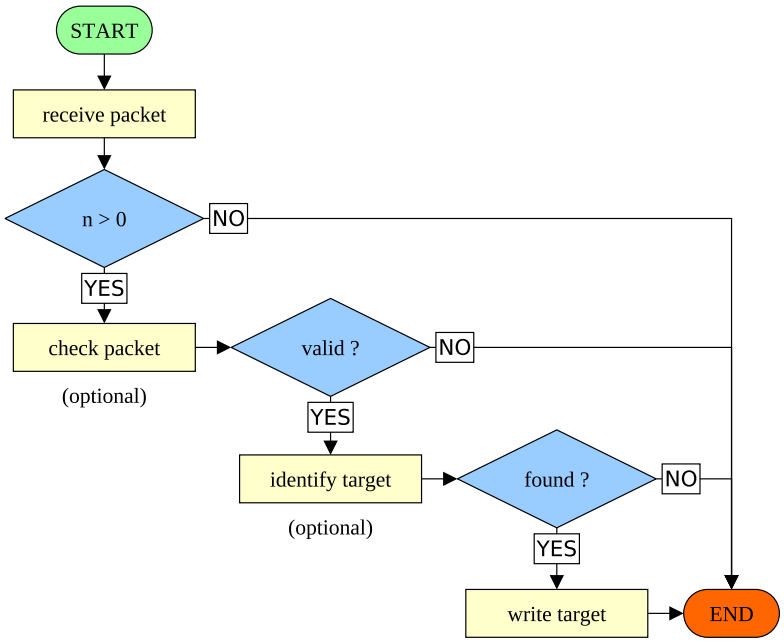

Figure 8. Example implementation of `handleData()`, ©*IRS*

D. Interfaces

So called interfaces are a means of buffering or forwarding data by the continuous handlers. The term might be a little misleading. It must not be confused with the model input/output interfaces. However, the interfaces described here are quite generic and can be configured for various applications.

The generic Interface class is depicted in figure 9. Interfaces are supposed to save, buffer, or forward information. Thus the base interface class provides a memory array, a size and a simulated address that shall not be confused with the physical address of the `Memory` variable.

An interface or derivatives of that class shall only be accessed via three public methods:

- `public int READ(uint, uchar*, uint, uint&)`
- `public int WRITE(uint, uchar*, uint)`
- `public void WriteMemory(uchar*, uint)`

Because an interface object can be accessed by multiple instances at the same time, these methods first check if the interface is not currently busy applying the modification by another object. If that is not the case they block the interface by themselves and call one of the respective protected methods:

- `protected int read(uint, uchar*, uint, uint&)`
- `protected int write(uint, uchar*, uint)`
- `protected void writeMemory(uchar*, uint)`

These methods are fully virtual and must be implemented by the inheriting classes.

`read(...)` is called whenever data needs to be retrieved from the interface. Depending on whether the derived interface is a Register, a Buffer or a RingBuffer `read(...)` manages the START and END markers of the saved data as well as the fill state of the used memory.

`write(...)` is usually called whenever data that is received from an external data source needs to be written to the interface. Depending on the used protocols and the device functionality, it performs all required data conversions and calls `writeMemory(...)` eventually.

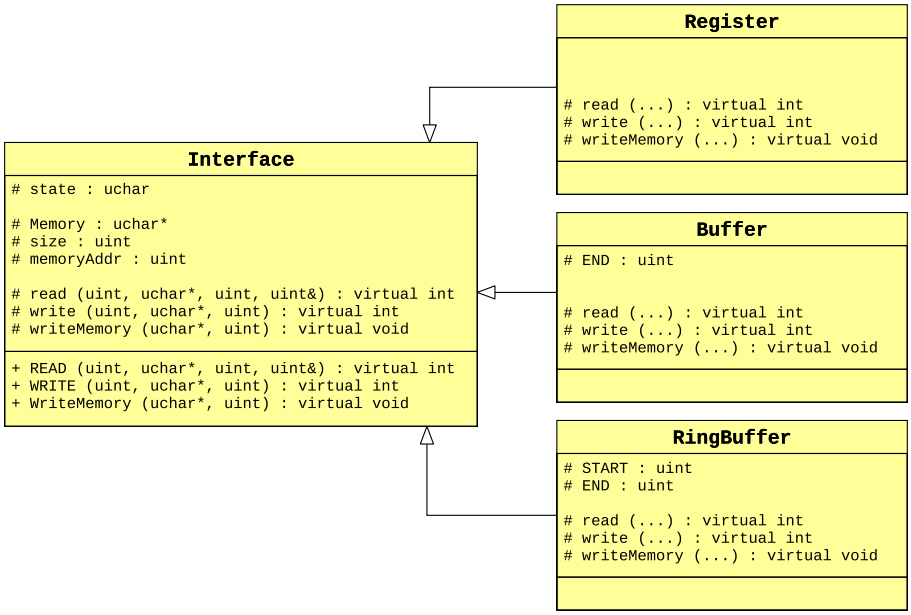

Figure 9. Generic Interface class and derivatives, ©*IRS*

`writeMemory(...)` is used to write an array of bytes to the memory of the interface. The method also manages the START and END markers of the saved data as it updates the fill state whenever data is written to the memory.

The derived classes Register, Buffer and RingBuffer already define all of these virtual methods, which is why further derivatives are supposed to inherit from one of these three classes. Depending on the device functionality and the application, derived classes can re-define these protected virtual methods. `read(...)` and `write(...)` for instance could also be used to access external data sources/sinks directly, without the need of a client. This is applicable e.g. when the external data source/sink is a specific input/output line of the model.

E. Client - Bottom Level

As indicated earlier, the client is the object that actually gets data from external sources as well as it transmits packets to the respective sinks. Depending on the test bench setup and the simulated model, the client can be configured for several kinds of sources/sinks. An overview about the selectable channels implemented in the RMAP[d] client used for *FLP* and *FLP2* is given by figure 10.

First alternative is the communication via some generic TCP socket. This channel mostly suits on-board computer models with the on-board software being hosted on a development board or a similar device which provides TCP communication. In this case the client must bind to the socket opened by the on-board software. *Flying Laptop* on-board software provides such functionality for developing and testing purposes [7]. Besides TCP one can think about implementing other transport protocols such as UDP, for instance. In this example one has to keep in mind that the accessed socket has to be considered as an external data source, although the on-board software is a part of the simulated on-board computer model. Data that has been received by the client in this example still needs to be processed by a continuous handler and forwarded by another client (as shown by figure 6). Same applies to the communication vice versa.

In case an on-board software is not hosted by real hardware but runs on something like a virtual machine, a different TCP communication channel can be selected. This is due to the fact that some commercial hardware emulators, such as the used *TSIM* simulator [8] by *Cobham Gaisler*, encapsulate the transmitted packages into their own protocol. Which is why the implemented client provides means of retrieving the

[d] *FLP* and its derivatives use *SpaceWire* and RMAP within their CDH. For convenience the following further explanations stick to that example instead of going through a generic description.

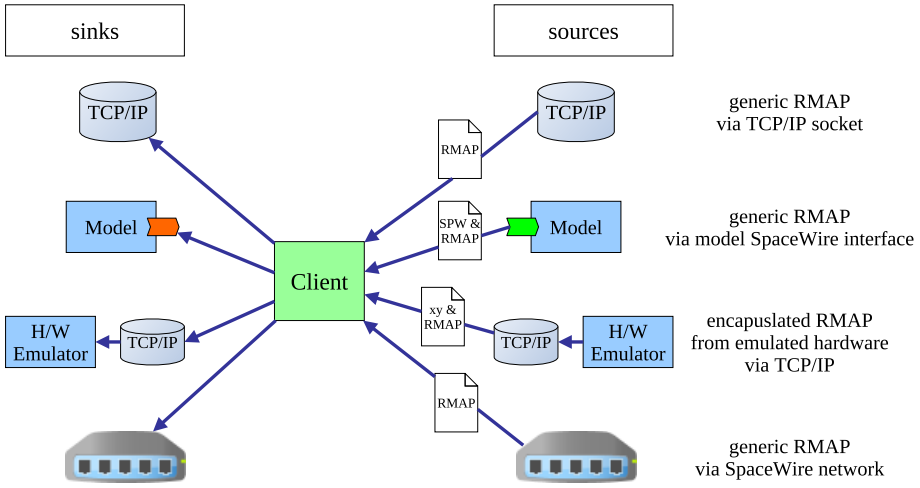

Figure 10. Different data sources and data sinks of the RMAP client, ©*IRS*

RMAP packages from such a proprietary protocol.

A third alternative is the reception of RMAP packages from a simulated *SpaceWire* line of the device model. This approach suits the simulation of almost any *SpaceWire* device assuming the model is connected to another model via such a simulated line.

For the case that the model needs to communicate with a real *SpaceWire* network, a fourth communication channel has been implemented. This applies to the case where the modeled device is one node in a *SpaceWire* network, but parts of the network are not simulated, as for example in the hybrid *FLP2* system test bench (see figure 3). For such a purpose, the developed RMAP client provides functionality to establish a connection to a *SpaceWire* network via a *4Links SpaceWire* interface [9], connected to the simulation computer via Ethernet.

An overview about the possible client configurations and the most suitable use cases is given by table 1.

Table 1. Overview of client source/sink configurations and applications

source/sink	received protocol	application
TCP socket	RMAP	connection to some sort of processor board
TCP socket	RMAP (encapsulated)	connection to emulated processor board
simulated SpW IF	RMAP	arbitrary SpW device model connected to other model
SpW network	RMAP	simulated SpW node in a real SpW network

Besides the pure routing functionality, the developed client provides an extensive implementation of the RMAP standard [6]. Thus it is capable of checking the received data according to the standard packet definitions. The client is furthermore simultaneously connected to a data source *and* a data sink. If the packet turns out being invalid, an error reply can be composed and sent right away. Yet, the packet checking as well as the sending of any kind of reply must be triggered by the continuous handler in charge of the client.

III. TESTING OF CDH MODELS

DEFINING tests of object-oriented software is always a major issue [10]. This is due to its complexity and the unpredictable states an object-oriented software might reach. The effort of qualifying an object-oriented software becomes even worse when it is multi-threaded or part of an interactive network. All of this applies to the described approach which extends the required effort to test these CDH models compared to other models of the simulator.

Although models are tested quite extensively, a 100 % coverage of any model state cannot be guaranteed, especially with limited personnel resources. Therefore tests have been developed that cover the most likely use cases of the respective models. However, the approach benefits from the fact that a lot of generic classes are re-used over a wide range of data-handling models. E.g. functionality of the RMAP client that has been verified in the scope of one CDH model test, doesn't have to be verified again for another model, because all CDH models in this satellite framework use the same client. Same applies to the interface class (see figure 9) and its derivatives, which are re-used as well. In this case only model specific derivatives have to be tested, e.g. in case they re-define virtual attributes or provide additional functions.

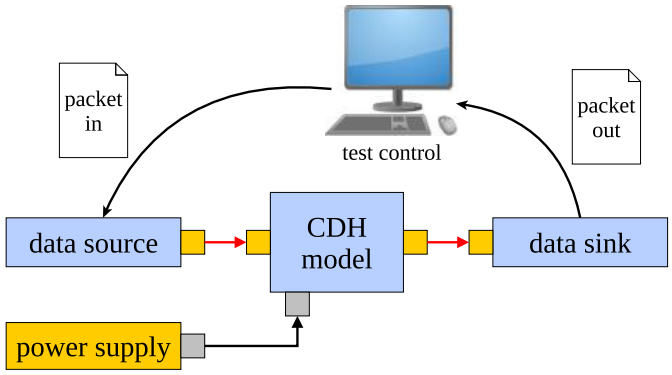

Figure 11. Setup for CDH model tests, ©*IRS*

Figure 11 shows a general setup of a CDH model test. First, the model is attached to a simple power supply model. This power supply provides the tested model with its nominal voltage in order to satisfy the power demand of the device. Providing the CDH device model with the correct voltage enables the continuous handlers (see figure 7) and thus the data processing by the model.

Then the model is fed with an input packet from a data source model. This data source model is a very simple model providing the same data interface the tested CDH model uses. Most likely the data is injected into a simulated *SpaceWire* line. The data source in this setup must not be confused with the data source the client accesses to retrieve data. In this setup the data source according to figures 6 and 10 is the simulated *SpaceWire* line. The model then must receive data from that line, process it and forward it as described in section II. This means that the implemented client needs to retrieve the transmitted RMAP packages from the *SpaceWire* line and the model handlers need to forward them. Afterwards the model responds upon the injected packets with some form of data injection into an output line by the same or a different RMAP client. Attached to the line where the device response is expected is a simple data sink model. The test is successful, when the output packet, received by the sink, matches the expectations stated by the test control sequence.

In order to fully qualify the client, a test like this must be repeated for all of the possible data source/sink configurations as specified by table 1.

IV. CONCLUSION

USING the described layered approach supports the simulation of significantly larger data networks within a synchronous satellite simulation than it used to be possible. This is because the introduced approach allows the simulation of CDH components where formerly real hardware was required. E.g. one can think about simulating routing devices instead of purchasing expensive hardware, if that suits to the purpose of the respective test bench. As stated in the first place, this allows to bind an on-board computer closer to a simulation and can therefore reduce the complexity of a hybrid STB.

Furthermore, the approach supports the simulation of an on-board computer itself. This is a quality that can taken advantage of e.g. when an entire constellation of satellites is simulated, instead of just one single satellite. Something like this is done when the focus of the STB lies not on the actual satellite system, but on the testing of aspects of constellation operations, such as constellation management. *Institute of Space Systems* is currently working on such a test bench dedicated to constellation operations research.

The concept is well supported by the implemented client, providing comprehensive connectivity to various data sources. This suits the development of several STB extension stages, where different data-handling components are either simulated or exist in form of real hardware. In such variable cases models can be configured easily for a different data source without the need of any rework. The fact that the implemented client only supports the remote memory access protocol[6] is yet a minor drawback.

First tests of CDH models implementing the described approach turned out quite promising. Hence, the introduced simulation methodology shows great potential for an extensive use within *FLP* and *FLP2* simulation projects as for future applications as well.

REFERENCES

[1] Manchaiah, D., Klink, A., Chintalapati, B., and Eickhoff, J., "A Dual-core LEON3 Computer Managing Smallsat Platform and Payloads," Tech. rep., Airbus DS GmbH, 88039 Friedrichshafen, Germany, 2018.

[2] Eickhoff, J., *The FLP Microsatellite Platform - Flight Operations Manual*, No. 978-3-319-23503-5, Springer-Verlag, Airbus DS GmbH, Friedrichshafen, Germany, 2016.

[3] Leidig, K. and Eickhoff, J., "Microsatellite Simulation for Constellation Research," *SciTech*, No. AIAA 2017-1553, University of Stuttgart Institute of Space Systems, Pfaffenwaldring 29, 70569 Stuttgart, January 2017.

[4] "ECSS-E-70-41A - Ground systems and operations - Telemetry and telecommand packet utilization," Jan 2003.

[5] "ECSS-E-50-12A - SpaceWire - Links, nodes, routers and networks," Jan 2003.

[6] "ECSS-E-ST-50-52C - SpaceWire - Remote memory access protocol," Feb 2010.

[7] Bätz, B., *Design and Implementation of a Spacecraft Flight Software Framework*, Ph.D. thesis, Institute of Space Systems University of Stuttgart, Pfaffenwaldring 29, 70569 Stuttgart, 2018.

[8] Gaisler, *TSIM2 Simulator User's Manual*, Kungsgatan 12, 411 19 Goeteborg, Sweden, June 2016.

[9] 4Links Limited, Suite EU2, Bletchley Park, Milton Keynes MK3 6EB, UK, *Diagnostic SpaceWire Interface*.

[10] Sneed, H. M. and Winter, M., *Testen objektorientierter Software: das Praxishandbuch für den Test objektorientierter Client-Server-Systeme*, Hanser, München, 2002.