Software tests and simulations for real-time applications based on virtual time

Martin Hierholzer, Geogin Varghese, Martin Killenberg

***Abstract*—Unit and integration tests are powerful tools to ensure software quality. Writing such tests for real-time applications accessing hardware requires not only replacing the real hardware with a virtual implementation in software. Also time must be controlled precisely. For a number of reasons the time scale in the simulated environment should not be identical to the real time: computations needed for a complex plant model might just be too slow for a real time simulation, or some long-term software behaviour should be tested in a short-running test. Communications with devices often require a specific timing which should be subject of a unit test. These examples demand using a virtual time scale in software tests.**

We present the VirtualLab framework as part of the MTCA4U tool kit. It has been designed to help implementing such tests by introducing the concept of virtual time and combining it with an implementation basis for virtual devices and plant models. The framework is designed modularly so that virtual devices and model components can be reused to test different parts of the control system software.

I. Introduction

THE VirtualLab framework allows testing control applications which access hardware without the need for dedicated testing equipment. The tests can be run on a standard PC, e.g. as a contineous integration test triggered by source code changes. VirtualLab has been developed as part of the MicroTCA.4 user tool kit MTCA4U[2]. Since the use of MTCA4U is not at all limited to MicroTCA-based applications, it is has recently been renamed into ChimeraTK[3].

In this paper, special emphasis is put on the concept of virtual time, which is a central element in the design of pure software-driven tests for real-time applications. With the understanding of this approach, the other concepts used in this framework can be explained. The final goal is to implement virtual devices which emulate any hardware components needed to run the application close enough to allow testing the application.

The tests itself can be written based on the BOOST Test Library[1]. A simple test would e.g. send an operator's command to the application, wait until the application is finished with the processing and then check if the virtual devices have received the right actions. The test can also induce a state change in one of the virtual devices, wait until the application has processed the information and check if it takes the right actions.

Virtual time - as opposed to real time - can be controlled by software. This is important since the tests are running outside the real-time environment. Simulating hardware components might take considerable CPU time and thus could require a slower execution than in the production setup. Also a typical contineous integration test server will run many jobs in parallel, thus no real-time behaviour can be expected from the application. Therefore, not only hardware components have to be replaced by virtual devices, also time should be substituted with virtual time while running the tests.

Manuscript received on June 1, 2016.

Martin Hierholzer is with Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, 22607 Hamburg, Germany (corresponding author, e-mail: martin.hierholzer@desy.de).

Geogin Varghese is with Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, 22607 Hamburg, Germany (presenter, e-mail: geogin.varghese@desy.de).

Martin Killenberg is with Deutsches Elektronen-Synchrotron DESY, Notkestr. 85, 22607 Hamburg, Germany (e-mail: martin.killenberg@desy.de).

II. Overall structure of a virtual test environment

Fig. 1 is showing the structure of a test using the VirtualLab at the example of a radio frequency (RF) control application for a particle accelerator. The working principle of such an application is of no particular interest here, it shall only be mentioned that a cavity, which is an accelerating element of a particle accelerator, is driven by a high-frequency electromagnetic field. This field needs to be regulated by a control loop running on a fast FPGA. The control application to be tested runs on a normal CPU, governs the many parameters of the control loop and extracts diagnostic information.

For testing this application, both the firmware running the control loop as well as the cavity have to be replaced with a model which is just close enough to the real hardware that the application can run properly. Some components present in the real setup can be neglected completely, such as the digital-to-analogue and analogue-to-digital converters and the high-power amplifier driving the cavity. Important is only that the software observes a realistic response to its actions in the firmware registers.

The various building blocks shown in Fig. 1 will be explained in the following sections.

III. The conecpt of virtual time

In a very simple approach, virtual time can be achieved by reimplementing any time dependent functionality based on a simple variable representing the current time. In case of the VirtualLab, a 64-bit integer has been chosen which represents picoseconds. This allows sufficient resolution even for high frequency applications like the RF controls of a particle accelerator, while still being able to reflect long times like several month.

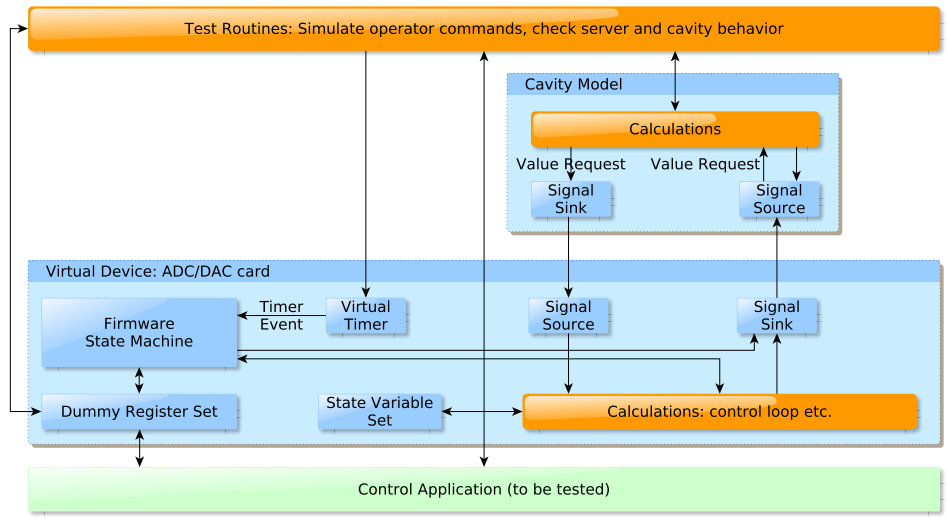

Fig. 1. Overall structure of a test environment written with the VirtualLab framework at the example of the test of the RF control application server of a particle accelerator. The control application server to be tested is represented by the green box at the bottom. It interacts with an ADC/DAC card which has been replaced by a virtual device for the tests. The virtual device is connected to a simple cavity model providing the ADC with a realistic response to the DAC signal. The control loop is running on an FPGA on the ADC/DAC card and the control application communicates through registers with the card's firmware. All virtual components can be controlled by the test routines, which also may interact with the control application server.

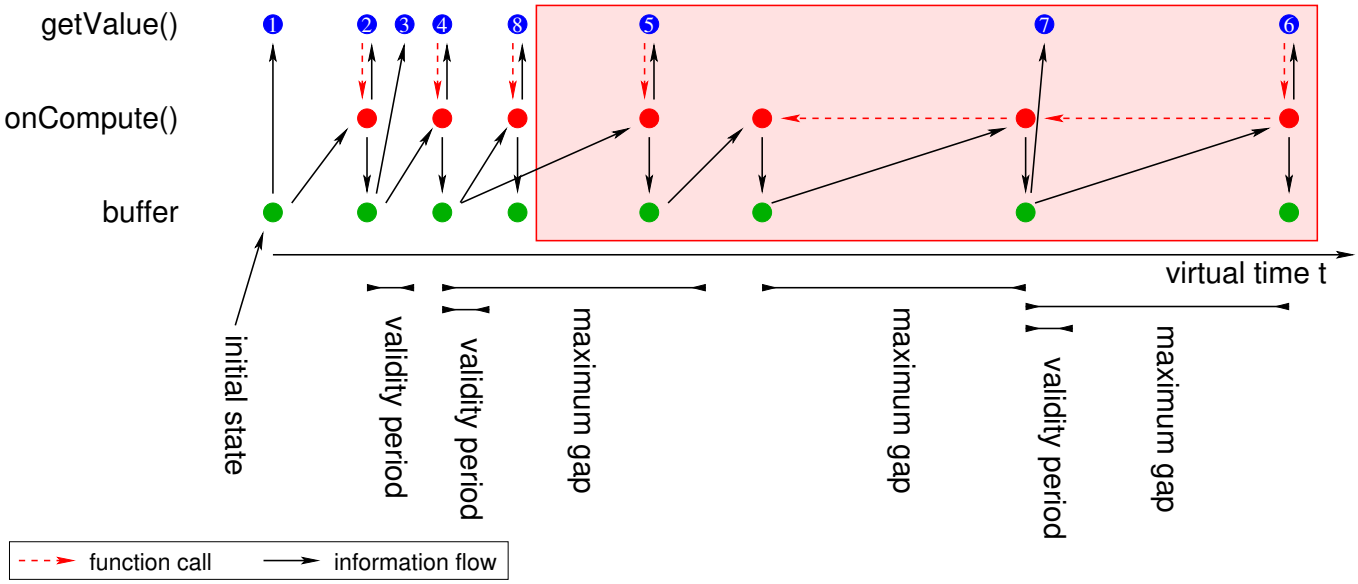

Fig. 2. Sketch of the implementation of virtual time in the class `StateVariableSet`. The blue circles on the top represent the requests via a call to `getValue()` incoming in the sequence of the numbers inside each circle. The red circles in the middle represent the calls to the callback function `onCompute()` which triggers computing a new model state. The green circles at the bottom represent the entries in the buffer storing the computed states for later re-use. New states are computed according to the configured validity period and maximum gap time. Request number 8 requires computing a new state in the past, which triggers the deletion of any later state which had already been computed (illustrated by the red box).

The replacement with virtual time should only be done for those parts of the application which actually need to be controlled during the tests. This usually includes any parts interacting with the (virtual) hardware, while e.g. some network communication routines might interally use timers which should be kept unchanged.

To save unnecessary computations especially in more complex setups, the VirtualLab framework does not assume a fixed sampling rate when stepping through virtual time to compute model states of the virtual hardware components. Instead, each component will request updates for a given virtual time from the other components when needed. This has a number of implications on the implementation of virtual time:

- Multiple components might need to request information from the same component. To save computing time, already computed states shall be reused.
- Often, a state cannot change significalty within a short period of time. Thus a certain *validity period* can be assigned, so a second request falling within this period will receive a copy of the state returned to the first request.
- A slowly updating component might request values from

a quickly changing component, which needs a certain minimum sampling frequency to compute proper values. Intermediate values must therefore be computed before returning the requested state. The maximum distance between those intermediate values is called the *maximum gap*.
- The virtual time is not the same for all components, e.g. due to simulated cable delays.
- Requests might come in for a virtual time earlier than the latest computed state, e.g. if some event triggered by the latest state results in an action requiring a more dense sampling of a period in the past.
- If states are computed in an approximation only (e.g. by numerically solving a differential equation), going back in time as described by the last point will "slightly" violate causality, because the recalculated values can differ. This needs to be taken into account while designing the model components and tests, and must be mitigated by other means. If the approximation is good enough, this effect often presents no issue.

The described behaviour has been implemented in the class `StateVariableSet`. Each model component which has a time-dependent internal state must use this class to store its state. The model provides a callback function named `onCompute()` to the `StateVariableSet` instance, in which new states will be computed.

The behavour of this class is illustrated in Fig. 2. In the setup phase, an initial state has to be provided. The first request comes in for the virtual time $t = 0$ and thus will be answered with the initial state. Request 2 will trigger the callback function which computes a new state based on the latest one available, which is the initial state. Request 3 falls into the validity period of the state computed on the second request and thus will receive the same state. Request 4 again needs to trigger the callback function, since it falls outside the latest state's validity period which was not prolonged by request 3. As long as requests stay within the maximum gap time w.r.t. the latest state, new states can be directly computed, as in request 5. Request 6 exceeds the maximum gap time by more than a factor of two, thus two intermediate states will be computed. Requests can also go back in time. If such a request falls within the validity period of an already computed state, like in request 7, that state will be returned. On the other hand, request 8 goes back in time but does not coincide with an already computed state, thus a new state has to be computed. In this case, all states which were already computed for a later virtual time will be deleted, since they are no longer valid. A second request at the same virtual time as request 5 might now return a slightly different result, since it would be based on the result of request 8 instead of 4.

IV. Signal sinks and sources

To allow reusing virtual devices and model components in different virtual setups, each component is kept separately as much as possible. Signals exchanged between the components are transported by signal sinks and sources, which can be connected to each other in the test routines e.g. during initialisation. As stated in the previous section, model states are computed upon request. This implies that also signal values need to be *requested* from other components (as opposed to pushing updates to connected components). To simplify the implementation of trivial components, a signal source already contains a `StateVariableSet` which takes care of calling the callback function for computing new values and storing them for reuse. As a welcome side effect, the model can also obtain its own past output values, if it needs them for its computations.

The signal source usually computes values upon request in the callback function. In addition, components can also actively provide values to their signal sources. This can be used e.g. if a virtual device receives values from the control application and has to provide them to other components.

V. Dummy register set

The virtual device is plugged in as a back end into the MTCA4U/ChimeraTK DeviceAccess library (see also [2]). The DeviceAccess library is normally used to access real hardware and allows transparently switching between several back end implementation to communicate with devices e.g. through PCI express or some network-based protocol. This mechanism can also be used to replace the hardware with a virtual implementation in software. VirtualLab provides a base class `VirtualLabBackend` with useful utilities to implement such virtual devices more easily. Devices in the DeviceAccess library are register-based, which means they define several named registers consisting each of a single value or an array of values.

VI. Firmware state machine

To implement the firmware of virtual devices, a state machine based on the BOOST meta state machine library[4] is used. State machine events will be generated when writing or reading registers by the control application, which can be used e.g. to trigger model computations by requesting values from the device's signal sinks.

VII. Virtual timers

Firmware often needs to perform periodic actions, like reading samples from an analogue-to-digital converter and writing them into a buffer. For such tasks, virtual timers can be set which will fire state machine events when they are expiring in virtual time. Multiple virtual timers can be combined to count time consistently, e.g. when using one timer to trigger the individual samples and a second timer to trigger sampling a short block of samples periodically. When advancing the combined timer until the slower second timer will fire once, the faster first timer will be automatically fired many times to keep them consistent. The timers can be controlled by the test routines and should generally be kept synchronised to the virtual time of the application to be tested.

VIII. Interaction of all components

The interaction of all VirtualLab components can be explained best at the example of the RF control application as

shown in Fig. 1. The control application communicates with the virtual device in the same way as with the real device in the production setup. It also interacts with the rest of the control system (e.g. an operator panel), which is here emulated by the test routines. The firmware state machine of the virtual device governs the control loop by updating its parameters when the control application changes the control registers. It also gathers diagnostic information, like a recording of the RF amplitude measured in the cavity over time, and provides it to the application in an array register. The control loop is acting on the cavity model and thus is connected through signal sinks and sources to it. Since the control loop implements also an integral controller, it needs to store an internal time-dependent state in a `StateVariableSet`. The test routine is able to control the virtual time of the firmware through a virtual timer. Since the firmware fills the diagnostic buffers with information from the signal sink, the virtual timer effectivly controls the virtual time of the control loop and the cavity model as well.

The test routine first initialises the VirtualLab environment by instantiating the virtual ADC/DAC card and the cavity mode, and connecting their signal sinks and sources. Next it starts the control application in a separate thread and allows it to initialise itself. Once the application is initialised, the test routine can start the actual tests.

Let us assume a procedure to ramp up the RF amplitude should be tested. The control loop implemented in firmware shall only be capable of keeping the RF amplitude at a given setpoint, so the control application has to increase the setpoint in small steps until the desired amplitude is reached. The control application shall receive periodic interrupts from the firmware at which the hardware interactions should take place.

First, the test routine sends the operator's command to the control application to ramp up the amplitude to a certain value. Then it advances the virtual timer until the interrupt is triggered. The application reacts to the interrupt by writing the setpoint of the first step to the register set. The firmware updates the control loop parameters accordingly. Now the test routine again advances the virtual timer until the next interrupt triggers. Since the firmware records the measured RF amplitude into a buffer (for later readout by the application), it requests values from the signal sink for each sample in the buffer. This is realised by combining two timers, one to trigger recording a sample and a second to generate the interrupt. The necessary calculations for all samples for the cavity model and the control loop are triggered intrinsically by the respective signal sources, obeying the configured validity periods and maximum gap times of the signal sources. The test routine checks now if the desired setpoint has been reached. The control application receives the next interrupt and reads the sampling buffer. It will do some safety checks and proceed with the next amplitude step.

The test routine repeats these steps until the final setpoint has been reached. This will test a successful ramping up of the amplitude. To test error conditions, it repeats the entire procedure but induces a fault at some point, e.g. let the amplitude drop to zero by instructing the cavity model to return constantly zero. In this case the control application stops the ramp, changes the setpoint to zero and informs the operator with an error message. All these actions are checked by the test routine.

IX. Conclusions

The VirtualLab framework allows testing real-time control applications in a non-real-time environment without the need for dedicated testing hardware. Virtual time is introduced to facilitate control over timing aspects of the virtual hardware and the application to be tested. The framework is available as open source software[5] under the GNU Lesser General Public License. It integrates well with the other components of the MTCA4U/ChimeraTK framework, especially the DeviceAccess library.

References

[1] *BOOST test library documentation*, http://www.boost.org/doc/libs/1_61_0/libs/test/doc/html/index.html.

[2] N. Shehzad et al., *Modular Software for MicroTCA.4 Based Control Applications*, *These Proceedings*, 20th IEEE Real Time Conference, Padova, Italy, 2016

[3] *ChimeraTK - Control system and Hardware Interface with Mapped and Extensible Register-based device Abstraction Tool Kit*, https://github.com/ChimeraTK

[4] *BOOST meta state machine library documentation*, http://www.boost.org/doc/libs/1_61_0/libs/msm/doc/HTML/index.html

[5] *Source code of the VirtualLab framework*, https://github.com/ChimeraTK/VirtualLab