ORNL/TM-2017/45

Simulation-based Testing of Control Software

Ozgur Ozmen
James Nutaro
Jibonananda Sanyal
Mohammed Olama

**Approved for public release.
Distribution is unlimited.**

February 10, 2017

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2017/45

Computational Sciences and Engineering Division

Simulation-based Testing of Control Software

Ozgur Ozmen
James Nutaro
Jibonananda Sanyal
Mohammed Olama

Date Published: February 2017

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

ACRONYMS

ADEVS	A Discrete Event System Simulator
COCOMO	COnstructive COst MOdel
CBC	Central Baptist Church
F	Fahrenheit
K	Kelvin
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
HVAC	Heating, Ventilation, and Air Conditioning
LBNL	Lawrence Berkeley National Laboratory
QEMU	Quick Emulator
UI	User Interface
ORNL	Oak Ridge National Laboratory

ACKNOWLEDGMENTS

Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.

ABSTRACT

It is impossible to adequately test complex software by examining its operation in a physical prototype of the system monitored. Adequate test coverage can require millions of test cases, and the cost of equipment prototypes combined with the real-time constraints of testing with them makes it infeasible to sample more than a small number of these tests. Model based testing seeks to avoid this problem by allowing for large numbers of relatively inexpensive *virtual* prototypes that operate in simulation time at a speed limited only by the available computing resources. In this report, we describe how a computer system emulator can be used as part of a model based testing environment; specifically, we show that a complete software stack - including operating system and application software - can be deployed within a simulated environment, and that these simulations can proceed as fast as possible. To illustrate this approach to model based testing, we describe how it is being used to test several building control systems that act to coordinate air conditioning loads for the purpose of reducing peak demand. These tests involve the use of A Discrete Event System Simulator (ADEVS) and Quick Emulator (QEMU) to host the operational software within the simulation, and a building model developed with the MODELICA programming language using Buildings Library and packaged as a Functional Mock-up Unit (FMU) that serves as the virtual test environment.

1. INTRODUCTION

Butler and Finelli [1] discussed the problem of testing for the high reliability of software and concluded that, because of the cost and time required to execute a test, it was impractical to demonstrate that a software system will exhibit a given mean time to fail. To illustrate the problem, they offered a simple model of the testing time required to demonstrate a mean to fail μ. The testing protocol involves creating n copies of the system to be tested and allowing these to run until $r \leq n$ are observed to fail. Larger r and n increase the statistical significance of the test. The required test duration D is

$$D = \mu \frac{r}{n} \tag{1}$$

Focusing on just the types of building control systems discussed in this report, we might wish for the mean time to failure of the control software to be greater than the mean to time failure for the electrical power. A justification for this choice is that a power failure will cause the software system to restart, effectively resetting its age to zero (i.e., by restoring the software's state to what it was at installation). A typical utility customer will experience 1 - 2 power outages per year, so let us assume a mean to time fail of 6 months. Demonstrating this mean time to failure would require at least sixth months if we have a single test facility (i.e., a building that can serve as a dedicated test site), and our confidence in the estimated failure rate would be very poor. The testing time can be reduced by purchasing more buildings (i.e., increasing n), but this would be impractical in almost every case. On the other hand, a six month test and a fixed cycle is impractical for the type of low cost systems considered in this article. As did Butler and Finelli, we can conclude that testing for a desired level of reliability is impractical.

If, on the other hand, n could be made as large as we like then the testing time can be made arbitrarily small. Specifically, if we can host the software as it will be deployed in a simulated building, then the number of replicates is limited only by the availability of computing power. Moreover, if the computer system on which the software will be deployed is less capable than the computer system that will host the simulation, then it may be possible to execute our tests faster than real time and so further compress the test and fix cycle. For the sake of illustration, let us assume that one hour of testing can be accomplished with one hour of simulation (i.e., simulation time to real time is one to one). Modern cloud computing services can provide one hour of computing time for roughly $1, and so the six months of testing described above could be done for less than $5,000. For the cost of a single, physical test facility, it could be possible to purchase years, and possibly decades, of simulation based tests; in this case, *n* would be chosen to provide a practical, real world time to complete the tests.

To demonstrate that this vision is technically feasible, we have leveraged the QEMU [2] computer system emulator and, by making suitable modifications to how time is managed by its simulated computer hardware, encapsulated this emulator within a discrete event model implemented in the ADEVS [3] discrete event simulator. This discrete event model of a computer system hosts the control system as it is deployed in the field. We have created a virtual test building using the Modelica modeling language and compiled this into an FMU [4] that is loaded into the ADEVS simulation system. By running real control software inside of the simulated building, we are able to execute a number of test cases limited only by the availability of computational equipment. We demonstrate this simulation based testing infrastructure with three scenarios: *(i)* a model of the Central Baptist Church (CBC) Building with 4 roof-top Heating, Ventilation, and Air Conditioning (HVAC) units and control software that operates these units to limit peak power demand; *(ii)* the same model of the CBC Building with a VOLTTRON [5] based control, and *(iii)* orchestration of multiple buildings via VOLTTRON for the purpose of following a power regulation signal. In the next section, we will describe time managed computer system emulation that enabled our technology.

2. TIME MANAGED COMPUTER SYSTEM EMULATION VIA QUICK EMULATOR

QEMU is a computer system emulator that can execute real software on simulated hardware. To emulate a computer system, QEMU dissects the instruction sequence of the software to be executed into chunks called translation code blocks. These blocks are translated into instruction sequences for the computer on which QEMU is executing by using a just in time compiler similar to those widely used by programming languages like Java and Python (QEMU also has a direct execution mode that skips the translation step, but we will not consider that mode here). QEMU also simulates hardware that is attached to the emulated processor; this attached hardware includes memory, system buses, timers and clocks, and equipment such as disks, network cards, video cards, mice, keyboards, and serial ports.

An important feature of QEMU is its *icount* mode in which the passage of time perceived by software on the emulated computer is linked to the number of instructions that the emulated computer has executed. The effect of this mode is to advance simulated time by a fixed (usually 1) number of nanoseconds for every emulated instruction that is executed. A second related feature is the QEMU virtual timer. This timer allows for scheduling events that will occur at some future value of the simulation clock in precisely the same manner as in a discrete event simulation. When the event time is reached, QEMU stops execution of the emulated computer and invokes an event handler. Execution of the emulated computer resumes when the event handler returns.

The *icount* mode can be configured such that when the emulated computer has no work to do, the simulation proceeds forward in time to the next event in the event list of the virtual timer. Hence, when the emulated hardware is idle, QEMU proceeds just as in a discrete event simulation by making instantaneous jumps through virtual time. (This mode is enabled with the command line option $-icount$ 1, $sleep = off$). Following the use of these same features closely by QBox [6] to synchronize the virtual clock within QEMU with the simulation clock in a SystemC model, we exploit the icount execution mode and virtual timer to use QEMU as a component in an overarching, constructive, discrete event simulation.

Upon startup, the modified QEMU software attaches to a UNIX domain socket, which we will call the synchronization socket, that has been created by the overarching simulator. QEMU waits for a time advance value to be written to this socket, and then schedules a virtual timer event for that interval into the future. The emulator then executes instructions until that virtual timer event occurs. When the event happens, QEMU writes a value equal to the simulated time that has elapsed within QEMU. This process repeats until the simulation is terminated. The startup sequence and the advancement of time within QEMU and the overarching simulator are illustrated in the sequence diagram of Fig. 1.

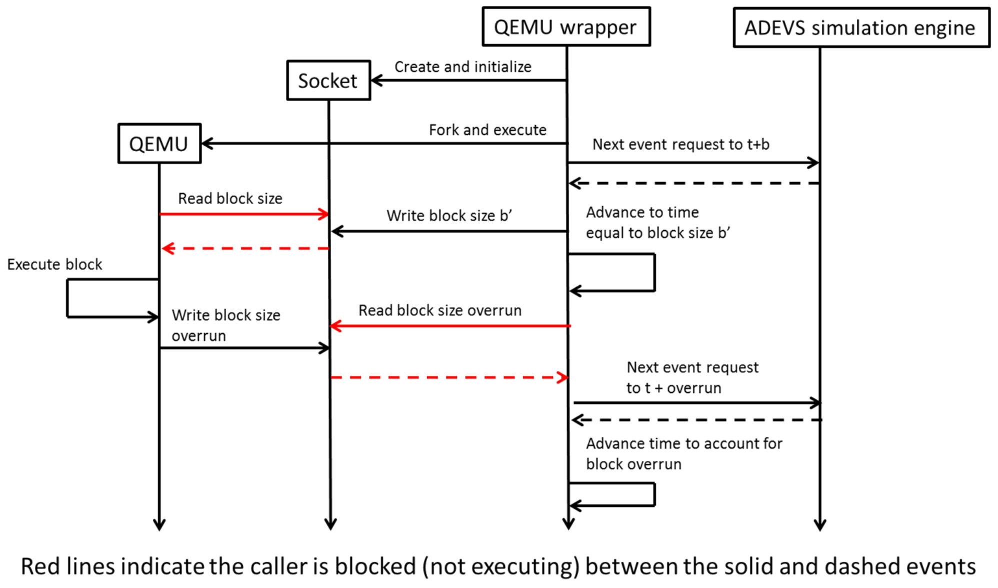
QEMU wrapper
ADEVS simulation engine
Socket
Create and initialize
QEMU
Fork and execute
Next event request to t+b
Read block size
Write block size b'
Advance to time equal to block size b'
Execute block
Write block size overrun
Read block size overrun
Next event request to t + overrun
Advance time to account for block overrun
Red lines indicate the caller is blocked (not executing) between the solid and dashed events

Fig. 1. Time synchronization between QEMU and the overarching simulator

The software executing on the emulated computer can interact with other models via the emulated serial port and network interface card, which are provided with QEMU. Communication between the discrete event simulator and the QEMU device model also occurs through a UNIX domain socket. This method of communication is an intrinsic part of the QEMU device models. During the execution of a time advance by QEMU, devices may write data to this socket. In the case of a network card, these data are complete network packets sent by the software executing on the emulated computer. In the case of a serial port, these data are sequences of characters written to the serial device by the software running on the emulated computer. When QEMU completes its time advance, the overarching simulator reads these data from the appropriate UNIX domain sockets and it writes to these sockets any packets or characters that have been transmitted to the emulated computer.

If large amounts of data are to be exchanged between QEMU and the overarching simulator, it is possible that the buffers backing the UNIX domain sockets will become full and cause the sender to block. If this happens, time will cease to advance if the intended recipient is waiting for a time advance to complete before reading from the socket. This potential problem is eliminated by having the overarching simulator dedicate a thread to reading and writing each socket. The reading thread loops on a blocking read of the socket. As messages become available, these are placed into a queue from which the overarching simulator will extract them at the next synchronization point. The writing thread accepts data to be sent to the QEMU device and queues this for transmission. It writes queued messages to the socket using a blocking write. In this way there is always a thread available to extract data from the buffer and allow QEMU to advance in time, and to write data to the buffer while the simulator advances.

Timing errors in this synchronization protocol can result from two sources. One source of error is the timer implementation in QEMU. The virtual timer will execute its events at some instant following the expiration of the event time stamp. Hence, the deadline could be missed and the elapsed time reported by QEMU at the end of the time advance may exceed what is expected by the overarching simulator. If this happens, the error is corrected by scheduling a catchup event in the overarching simulator for the reported overrun and then allowing QEMU to execute again after the catchup event has occurred.

The other source of error is artificial delay of messages exchanged with the emulated computer. With the proposed scheme, data is exchanged at the points of synchronization between QEMU and the overarching simulation. A necessary consequence of this approach is that messages will experience a delay of, at most, a single time advance plus the overrun described above. This error could be further exacerbated by real delays imposed by the UNIX domain sockets and the scheduling of QEMU threads that monitor those sockets. If these real delays are longer than the real time required to execute a time advance, then the difference will cause an undesired delay in the simulation equal to the number of time advances that elapsed while I/O operations were completed. A rigorous characterization of these potential errors is a subject for future research. In the next sections, we will demonstrate test scenarios of time managed computer system emulation.

3. TEST SCENARIO #1

In this test case, we have leveraged control software developed and deployed in previous work [7]. Our former work proposed a supervisory control strategy for limiting peak power demand by small and medium commercial buildings while still meeting the business needs of the occupants. The control strategy is designed to operate with building equipment, such as air condition and refrigeration systems, as they are presently installed in most small and medium commercial buildings. The control software acts as a supervisory management layer examining requests of individual building equipment controls and deciding which one to allow while satisfying a limit on peak power demand. This software was installed on a tinycore Linux image, which starts the control when it boots. Significant effort went into creating tinycore Linux images that could host the the control software; this work included dynamic re-partitioning of the tinycore Linux installation, compiling required C compilers in tinycore, as well as modifications to the system's library loading routines. These efforts resulted in a system image that is less than 200MB on disk and that can run in as little as 256 MB RAM while enabling networking and serial ports.

Leveraging ADEVS' FMU wrappers, the control software interacted with a building model developed in Modelica using the Lawrence Berkeley National Laboratory (LBNL) Buildings Library [8]. The building model is based on the gymnasium building in which the control software was originally deployed and tested. The physical layout of this building is shown in Fig. 2. The gymnasium has 4 HVAC units, each cooling the zone in which its thermostat is installed. Fig. 3 and Fig. 4 show the corresponding building models in the Dymola [9] user interface (UI).

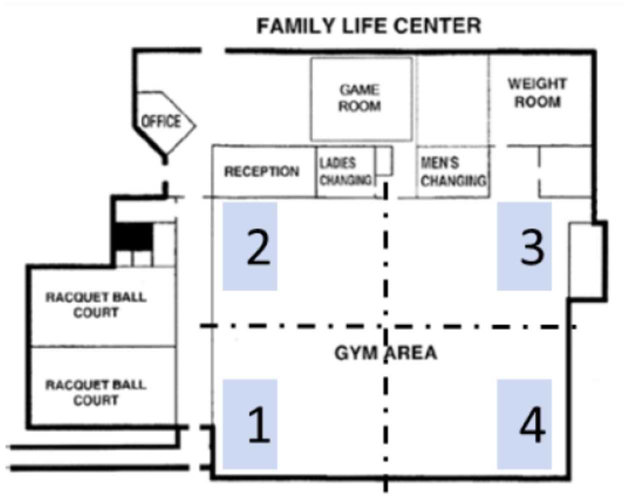

Fig. 2. Floor plan for the gymnasium (CBC Building). Dotted lines in the gym show the HVAC zone boundaries

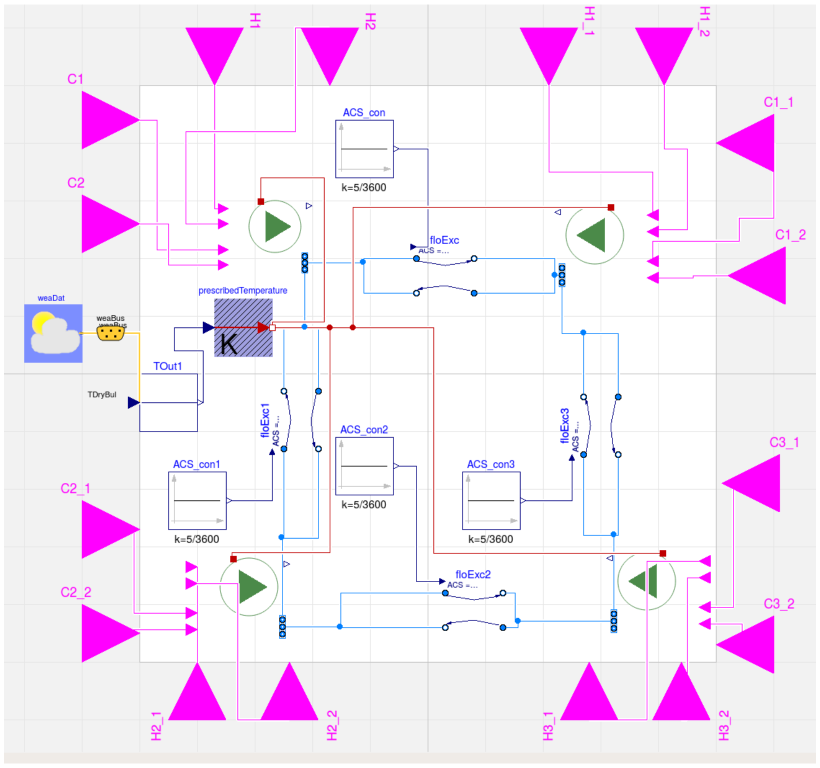

Fig. 3. Four zone building model each having their own HVAC unit and connected via air flows between the zones. Outside temperature is an input from a weather file for Chicago (USA_IL_Chicago-OHare.Intl.AP.725300_TMY3). Relay information (pink inputs) are input from the control software

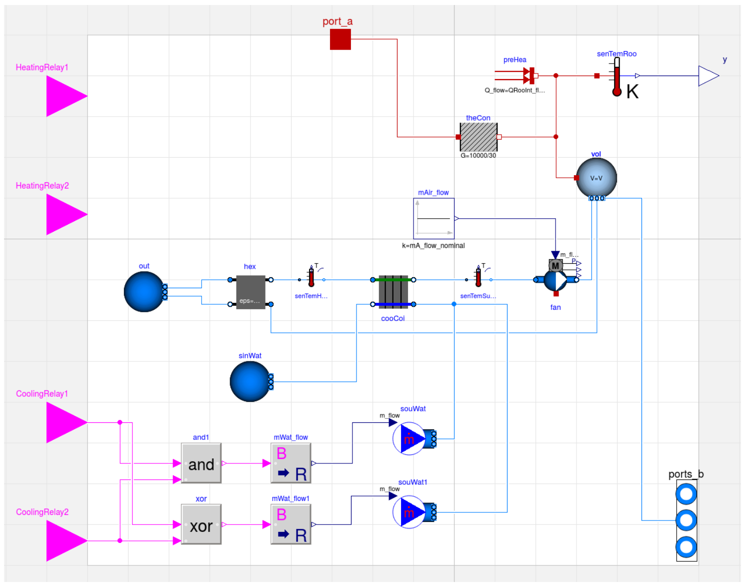

Fig. 4. A Zone model, which has mixed air, water cooling system, and control conditions, receives the relay information from upper level (pink inputs)

The control software communicates with thermostats in the real building by using the MODBUS protocol. To mimic this interface in the modeled building, we developed a mapping code leveraging libmodbus [10] and the Linux socat command. Specifically, we start two bidirectional virtual serial ports for byte streaming via socat (see Appendix A2). By adding a MODBUS server on one side of the ports, we are able to write and read messages to and from the MODBUS server. We write stream of bytes received from the control software (via the QEMU serial port) to the MODBUS server and use libmodbus to decode those messages. Upon decoding the message, the model constructs a response using the appropriate model parameters and then uses libmodbus to encode this response and send it to the emulated serial port. Fig. 5 represents the couplings of the test case along with the the directions of byte streaming.

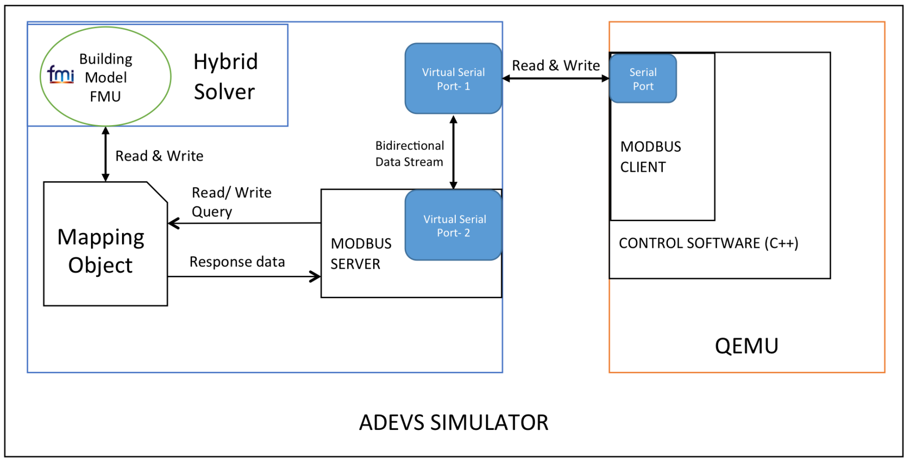

Fig. 5. Couplings of the components. Boxes represent wrappers and MODBUS contexts. Read and Write messages are non-blocking

Fig. 6 shows the room temperatures reported by the simulated thermostats during 12 hours of simulated operation, which required approximately 140 minutes real time to finish using a synchronization period between the emulator and the simulation of 1/10 seconds. Room temperatures in our test conditions start from a fixed value and temperature set point is 298 Kelvin (K) degrees.

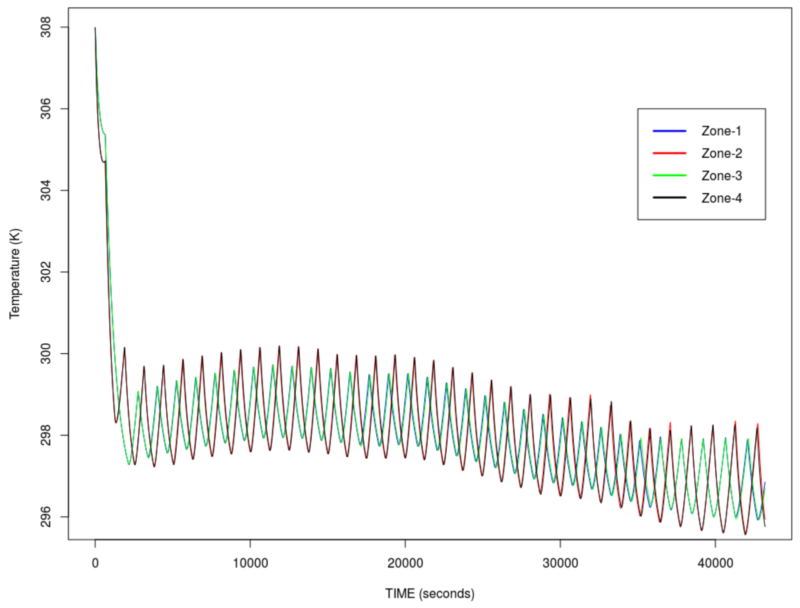

Fig. 6. Room temperatures of different zones evolve around the set point temperature over time

4. TEST SCENARIO #2

We have used the building model described in the previous section to demonstrate testing a control algorithm implemented as a VOLTTRON agent. The tiny core image described above was revised to host the VOLTTRON platform and necessary Python dependencies. The control in VOLTTRON implemented a simple thermostat logic. It monitors the temperatures reported by the thermostats. If the temperature of a zone is within ±1 degrees K around the set temperature (298 K) the the control turns OFF the HVAC unit; the unit is turned ON otherwise. Fig. 7 illustrates the test case and Fig. 8 shows the temperatures of each zone evolving over time. The room temperatures oscillate quite regularly due to the simple control logic. The 12 hour long simulation in Fig. 8 took approximately 10 minutes to finish with 1/10 second synchronization period for QEMU. In this test case, we used a simple data structure to directly send the temperature and command information without conversion to and from the MODBUS protocol. Hence, this test case executes faster than the previous test case, which was burdened with the overhead of the MODBUS encoding and decoding.

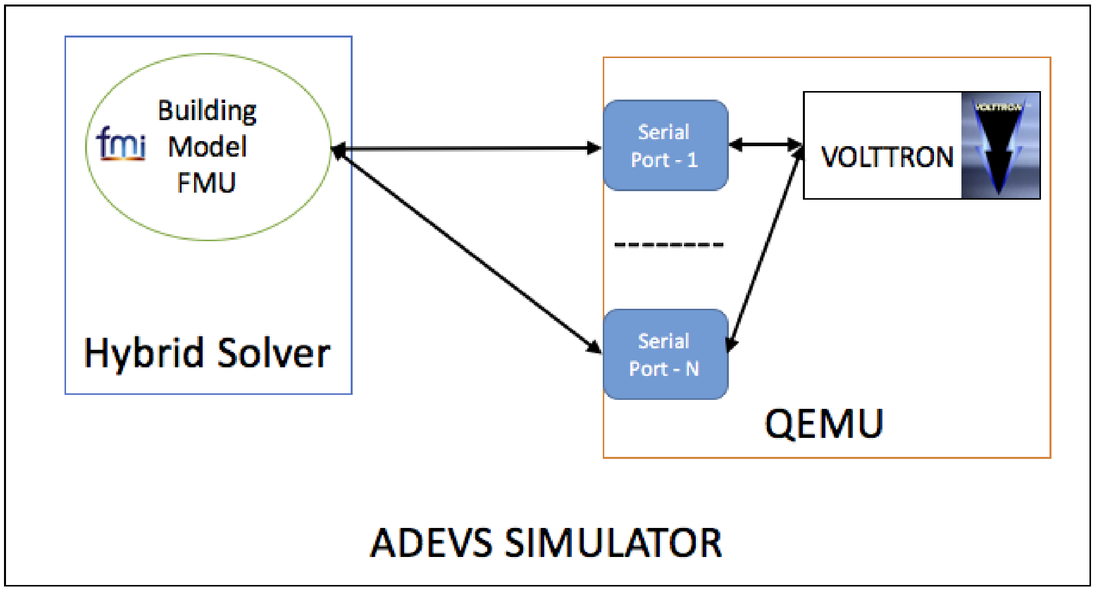

Fig. 7. Couplings of the components. Serial port N represents the number of HVAC units/zones controlled by VOLTTRON (4 in this test case)

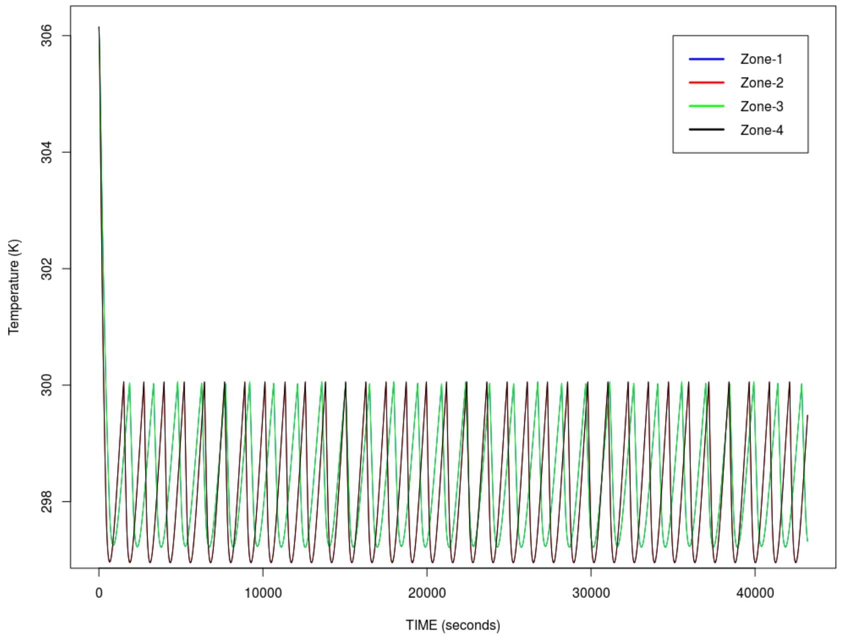

Fig. 8. Room temperatures over time for different zones oscillating around the temperature set point

5. TEST SCENARIO #3

In this test case, we leverage the implementation in the previous section and add one more layer to its control logic by developing two different kinds of VOLTTRON agents:*(i) Model Node* and *(ii) Master Node*. The model node gets temperature data from the simulated environment and sends HVAC relay information to the simulation via the QEMU serial port. The master node decides the relay settings (ON or OFF) for each model node based on a power regulation signal from the power utility company, temperatures in the simulated buildings, and set points of the HVAC units being controlled (for details of the control algorithm, see [11]). The master node interacts with the model nodes via the simulated network card. As the simulated environment, we used a collection of single zone buildings similar to that illustrated in Fig. 4. This one zone building model (Fig. 9) reacts to historical weather data (Chicago O'Hare International Airport, 2013) stored in an input file (the red box in Fig. 9).

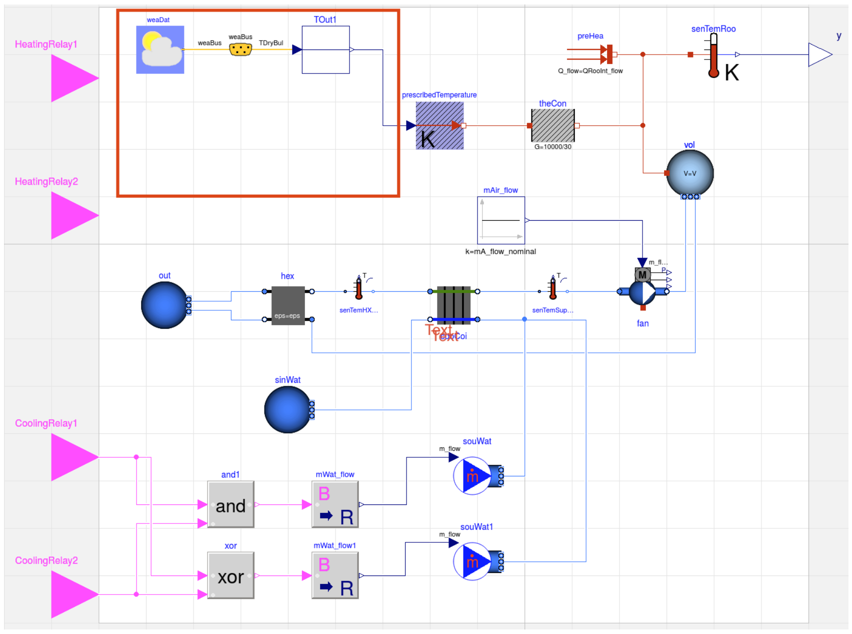

Fig. 9. One Zone Building model with the weather file, air conditioning system, and the control conditions. Relay decisions are received from the model node via pink inputs that are coupled with a serial port. Outside temperature is an input from a weather file for Chicago (USA_IL_Chicago-OHare.Intl.AP.725300_TMY3)

Fig. 10 shows the components of the test case. Each copy of the model node is coupled to its own simulated building; which talk to a single master node. Fig. 11 shows the evolution of room temperatures at each building. Overshooting of the temperatures are mostly due to the small number of HVAC systems and the amount of their power capacity dedicated as an ancillary service to the grid [11]. Based on findings in the previous study, [11] we expect to observe better performance from the control algorithm when the number of buildings is around 50. In this test case, we created 4 buildings due to the memory limitations of our desktop computer. We simulated 5 computers (incorporating the master node and 4 model nodes) and 4 building models allocating 500 Megabytes memory for each building. The real running time of a 12 hour simulated test was around 60 minutes.

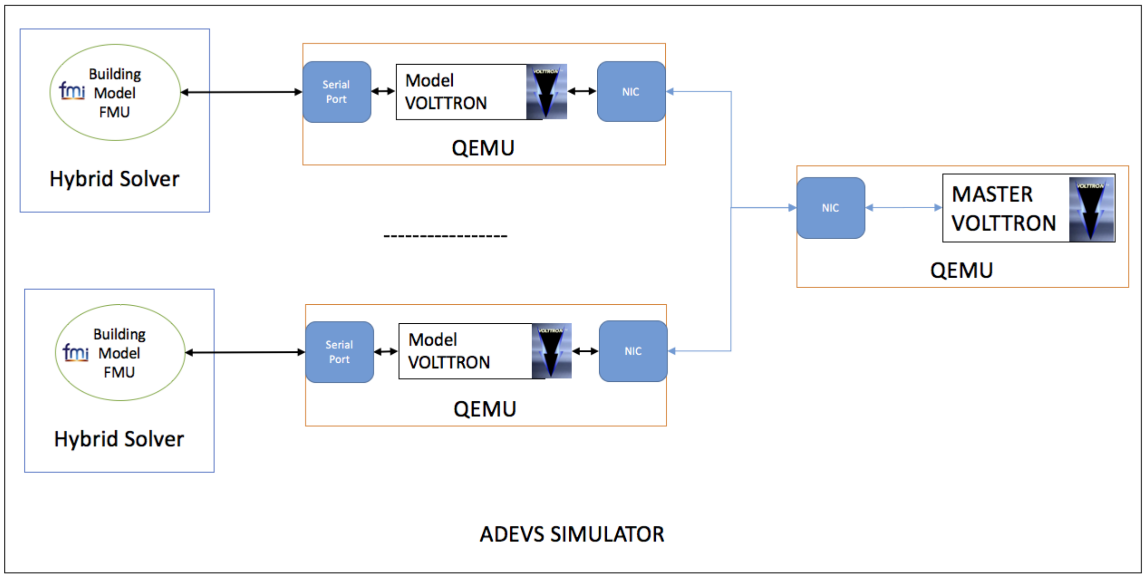

Fig. 10. Couplings of the components. Each model node represents a building and the master node monitors states of all model nodes and evaluates decisions

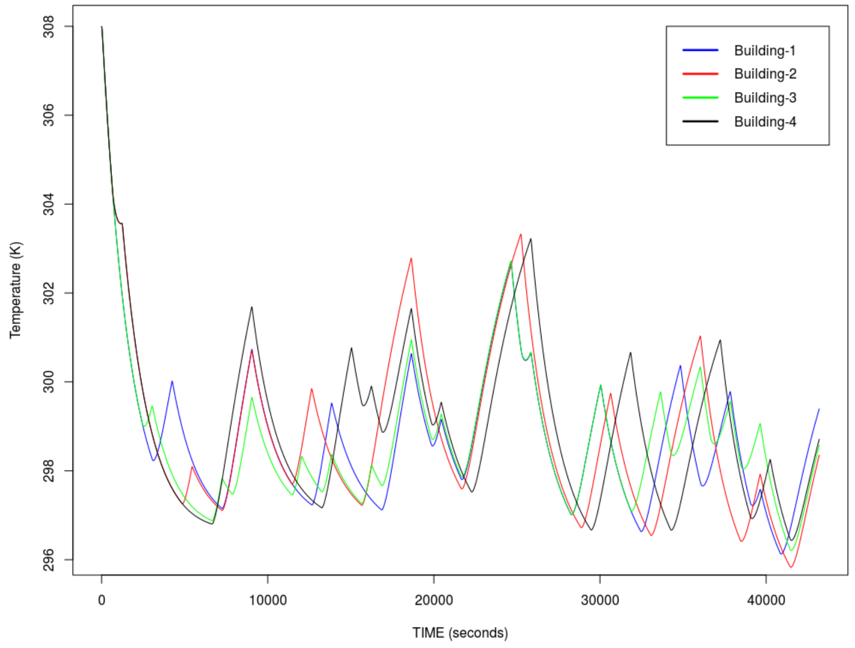

Fig. 11. Room temperatures of different buildings evolve around the set point temperature over time

6. DISCUSSION

In this report, we first demonstrate a test case that leverages our previous work and couples it to a model of CBC building developed in MODELICA programming language, which is shared as an FMU. This demonstration also shows the capability to use MODBUS protocol between control software and the simulator. Second, we replace the control algorithm in the first demonstration with a VOLTTRON based application as the control software. After this successful second demonstration of QEMU, VOLTTRON, FMU, and ADEVS integration, we demonstrate a test case with supervisory control algorithm that is developed as a VOLTTRON master node. Third demonstration also tests the use of virtual network cards for communication. These demonstrations show how a test-bed for more complex multi-building (community level) and multi-layer control algorithms can be realized with the new simulation technology. We leveraged a numerical solver for the buildings that uses h (step size) of 0.1 and provide synchronization of time between computers and simulation environment in every 1/10 seconds. Significantly, these demonstrations involved real software as it is, or would be, deployed in the field communicating with the HVAC thermostats through serial port (using MODBUS or adhoc protocols) and wireless (using network card) communication. In all cases, our demonstrations executed faster than real time.

By using larger synchronization time, faster solvers, and less frequent communication between emulators, we can achieve faster testing performance. With a more powerful computer (more memory and faster hard-drives), we can also achieve faster testing results for larger test cases. Additionally, leveraging parallel computing capabilities or by simply having embarrassingly parallel test cases, we can achieve thousands of times more testing time than real-time. This capability would enable feasibility for quantifying the reliability of more complex software by decreasing D (Eq. 1) significantly. Our future focus will be on testing merits of this technology in different problem domains against various software systems.

7. REFERENCES

[1] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 3–12, 1993.

[2] "Qemu website." http://www.qemu.org. Accessed: 2017-01-16.

[3] "Adevs website." http://web.ornl.gov/~1qn/adevs/. Accessed: 2017-01-16.

[4] "Fmi website." https://www.fmi-standard.org. Accessed: 2017-01-16.

[5] "Volttron website." http://transactionalnetwork.pnnl.gov/volttron.stm. Accessed: 2017-01-16.

[6] W. Mueller and F. Pétrot, "QEMU AND SYSTEMC," *1 st International QEMU Users' Forum*, 2011.

[7] J. Nutaro, O. Ozmen, J. Sanyal, D. Fugate, and T. Kuruganti, "Simulation based design and testing of a supervisory controller for reducing peak demand in buildings," in *International High Performance Buildings Conference, 2016*, 2016.

[8] "Buildings library website." https://simulationresearch.lbl.gov/modelica. Accessed: 2017-01-16.

[9] "Dymola website." http://www.3ds.com/products-services/catia/products/dymola/. Accessed: 2017-01-16.

[10] "Libmodbus website." http://libmodbus.org. Accessed: 2017-01-16.

[11] M. M. Olama, T. Kuruganti, J. J. Nutaro, J. Dong, and J. Sanyal, "Coordination and control of Building HVAC Systems to provide frequency regulation to smart grid," *Applied Energy*, Submitted: Jan, 2017.

[12] "Tinycore linux website." http://tinycorelinux.net/intro.html. Accessed: 2017-01-16.

[13] "Virtualbox website." https://www.virtualbox.org. Accessed: 2017-01-16.

[14] "Gnu wget website." https://www.gnu.org/software/wget/. Accessed: 2017-01-16.

[15] "Transfer.sh website." https://transfer.sh. Accessed: 2017-01-16.

[16] "Xcams website." https://xcams.ornl.gov/xcams/. Accessed: 2017-01-16.

[17] "Fmi downloads website." https://www.fmi-standard.org/downloads. Accessed: 2017-01-16.

APPENDIX A.TINYCORE IMAGE PREPARATION and CODE REPOSITORIES

APPENDIX A1. TINYCORE IMAGE PREPARATION

Tinycore Linux distribution is a unique and minimalist distribution of recent Linux kernel and operating tools [12]. We download the smallest image file (Core iso - 11MB) from tinycore website. Then using Virtualbox [13], we install the iso and add network capability to the installed virtual machine. Through network and internet connection, we install necessary packages and dependencies from TCZ repositories (*tce-load* commands). Additionally, we transfer other software packages that do not exist in the TCZ repository (i.e., libmodbus, control software itself) via wget [14] and transfer.sh [15]. When control software is tested and running as it is intended, we revise startup scripts (particularly .profile file) to set up serial ports, network authorizations and IPs, and to run the software automatically when the machine boots. After the virtual machine is ready, we convert it to .img raw file format and use it with QEMU.

APPENDIX A2. CODE REPOSITORIES

All demonstrations are shared in the repository below:

– https://code.ornl.gov/Sim_based_testing/demos.git

Sect. 3 code is in *CBC_qemu_modbus* folder, Sect. 4 code is in *qemu_CBC* folder, and Sect. 5 code is in *qemu_w_nic* folder. R script to generate plots of outputs is in *R* folder. Additionally, if image files used in these demonstrations are needed, they are in the repository below:

– https://code.ornl.gov/Sim_based_testing/images.git

Please contact Ozgur Ozmen (ozmeno@ornl.gov) if you have difficulties connecting to the repository or need to be granted access privileges to it. Users from outside of Oak Ridge National Laboratory (ORNL) can connect to the repository by creating an XCAMS account [16].

Users should make sure to compile ADEVS in advance and use FMI header files (for FMI function pointers) from FMI standard website [17] to be able to wrap FMUs in ADEVS. Also Makefiles should be revised with necessary PATH inclusions. README files in each folder give instructions on what each file does in that folder. When we run the first test case (Sect. 3), we enter Linux socat command below in Terminal and set up the virtual serial ports (i.e., /dev/pts/18) in ADEVS implementation:

– socat -d -d pty,raw,echo=0 pty,raw,echo=0