Hindawi
Scientific Programming
Volume 2024, Article ID 9648349, 13 pages
https://doi.org/10.1155/2024/9648349

Research Article

An Example of Modelica–LabVIEW Communication Usage to Implement Hardware-in-the-Loop Experiments

Massimo Ceraolo and Mirko Marracci

University of Pisa, Pisa, Italy

Correspondence should be addressed to Massimo Ceraolo; massimo.ceraolo@unipi.it

Received 1 November 2022; Revised 20 October 2023; Accepted 17 January 2024; Published 5 February 2024

Academic Editor: Cristian Mateos

Modelica is a very powerful language to simulate a very large set of systems, including electrical, thermal, mechanical, fluidic, control, and has already been used very extensively for several purposes, as the several Modelica conferences testify. Despite of this large literature, no paper seems to be available regarding the use of Modelica for real-time applications or hardware-in-the loop (HIL). This is a field where applications may be very fruitful. In this paper, the possibility of creating mixed software–hardware experiences (i.e., HIL), through combination of a Modelica program, the related simulation tool, a LabVIEW program, and the corresponding hardware is demonstrated. This demonstration is made using as an example a partial simulator of an electric vehicle running in a stand-alone PC, which communicates via User Datagram Protocol (UDP) packets with another PC running the LabVIEW program, which in turn is physically connected with the hardware-under-test. The obtained results are satisfying, given the inherent delay times due to the UDP communication.

1. Introduction

The Modelica language has appeared around 20 years ago as a general-purpose language for description of possibly large and complex models, including nonlinear time varying, containing discrete and continuous variables, etc. Modelica-based simulation tools can simulate different kinds of systems, such as electrical, mechanical, thermal control, etc., even simultaneously present as subsystems of a single comprehensive system. Several books [1, 2] and papers (all those presented at the Modelica conferences, listed in https://modelica.org/publications/articles), some of which by one of this paper's authors [3–5] describe characteristics, advantages, and applications of this simulation language, and they will not be repeated here again.

The flexibility of this language, which allows the corresponding programs to be used either for commercial (a good example of these is [6]) and free [7] simulation tools, makes it possible to use it for a lot, if not all, the simulation needs of most scientists and engineers.

Enlarging its capabilities will make this language even more attractive for existing and new users; that's where this paper can contribute.

Modelica is designed as a language for simulations, therefore is not explicitly designed for real-time operation. However, in relatively recent years the Modelica_DeviceDrivers library has appeared (presented in [8] and available for download from [9]), which allows to synchronise simulation and wall-clock time and to interact with users (e.g., trough keyboard, mouse, joystick), and external hardware (e.g., via serial port, User Datagram Protocol (UDP) [10], TCP/IP [11], CAN [12], etc.).

This can be exploited to realise simulation tools which interact with actual pieces of hardware, i.e., to create hardware-in-the-loop (HIL) experiments. Obviously, this is adequate only for relatively slow experiments since communication between Modelica tool and the environment outside introduce significant delays but is already very useful for many classes of the experiments.

For this paper, among the possible communication protocols, UDP was chosen, since it is much more flexible than the serial port, while still simpler and more manageable than TCP/IP; naturally, once the higher level architecture of the software is defined, people should in principle be able to

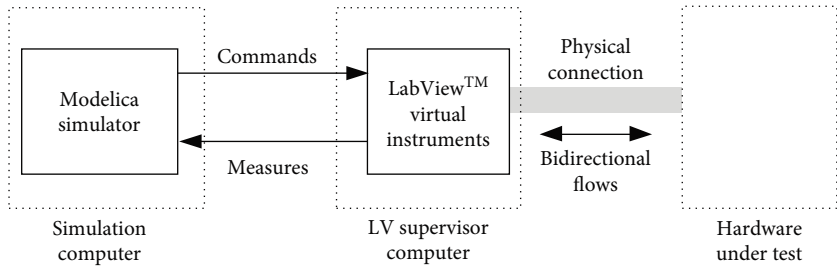

Figure 1: General arrangement for the Modelica–LabVIEW proposed experiments.

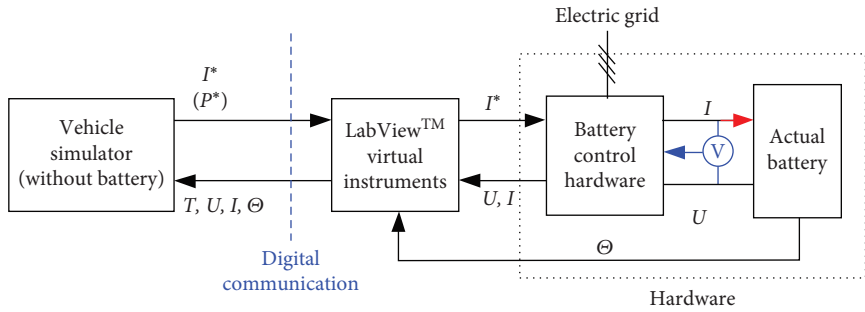

Figure 2: Arrangement of the HIL experiment in case the EUT is a BEV battery.

switch to different protocols with virtual no change on the lower level software layers.

In this paper we will consider the general architecture shown in Figure 1; a more detailed representation will be provided for the specific case of this paper's simulations in the next section.

The Modelica simulator sees the measures coming from the supervisor computer as they were taken from a simulated subsystem and elaborates them to generate corresponding commands.

This architecture has been detailed in this paper for a HIL experiment in which the piece of hardware physically tested is the battery of an electric vehicle, where the Modelica simulator simulates the vehicle and its dynamics. This is particularly interesting given the difficulty of accurately simulating the battery, which has a behaviour strongly nonlinear and subject to environment temperature, to ageing, etc.

The usage of National's LabVIEW [13] virtual instruments (VIs) in the supervisor computer was chosen because of its great flexibility, reliability, and speed of operation, and the availability of devices which allow to interface the user software with different pieces of equipment, including communication interfaces, such as serial ports and ethernet adapters. Its software allows to measure the quantities of interest on the battery and to control the instruments for managing battery charging and discharging in an integrated way.

Another interesting possibility to use Modelica-based simulators for HIL experiments is creating from the Modelica model a functional mockup unit (FMU), which can be integrated in the experiment control software, e.g., the a LabVIEW program; this further option will be dealt with in a subsequent paper.

2. Methods

2.1. The System in Case of Vehicle-Battery Experiments. As already mentioned, the architecture from Figure 1 is implemented in this paper for the case in which the equipment under test (EUT) is the battery of an electric vehicle, and the vehicle, and its interaction with driver and road, is software simulated.

To avoid undue complexity, the laboratory battery is here just as a single cell, representative of the actual vehicle battery which usually consists of series–parallel connections of the individual cells. The procedure proposed in this paper, however, is easily scalable to the larger batteries.

The diagram for this specific case is as shown in Figure 2. In this figure:

(1) I^* is the required battery current;
(2) T is the time at which the measures are taken;
(3) U is the measured battery voltage;
(4) I is the measured battery current;
(5) Θ is a temperature considered representative of the internal battery temperature map.

In the rare cases in which the simulated vehicle has some internal logic that requires to determine the vehicle behaviour also considering individual cells voltages and/or currents or different temperatures inside the battery pack, U, I, and Θ can be arrays of quantities. In this case, however, the increase in transmitted data can induce significant additional delays in the communications.

The simulator and the lab equipment can be physically far from each other, since the communication can be routed

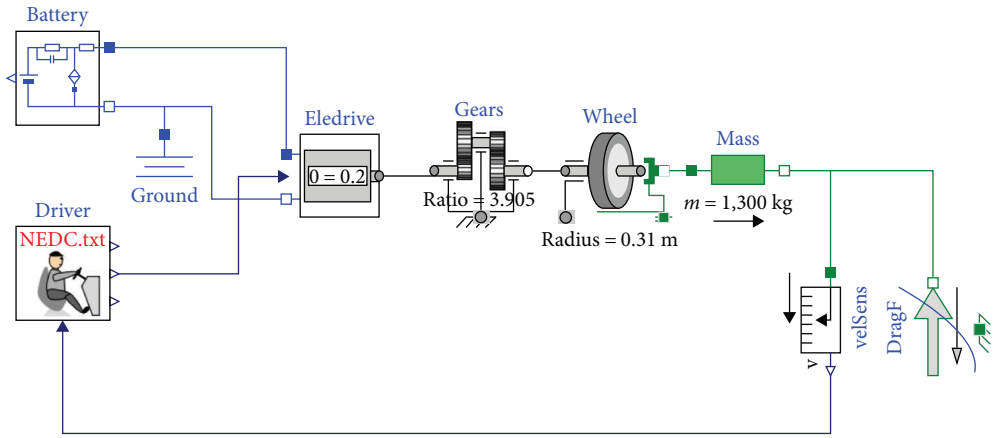

Figure 3: The Modelica diagram of the electric vehicle fully simulated in a Modelica tool.

through the Internet, in the cases in which corresponding delays are compatible with the experience to be carried out.

The vehicle simulator simulates the vehicle behaviour when subject to a specific test cycle, e.g., the NEDC [14] or the WLTC [15], and its behaviour is therefore dependent on its mass, mechanical (friction and aerodynamic) drag, and inertia forces. Some details in the following section. The vehicle simulator does not have a battery model inside; instead, it computes the requested traction power P^*, and requires it to the hardware. As an alternative, it may require the wanted current I^*; the latter solution is just a bit trickier, since the traction needs imply some power to be supplied, and this power translates into a current only when the battery voltage is known. Nevertheless, it is easy to implement, and was chosen for the remainder of this paper.

LabVIEW sends this request to the battery control hardware, and measures back the voltage U, current I, and cell's temperature Θ.

2.2. The Modelica Program. To understand the Modelica program used in this paper, let us first illustrate a diagram that does not use any HIL, since it has a battery model of its own, as illustrated in Figure 3.

Let us explain the graphical elements (submodels of the simulation model in figure) shown.

(1) Driver: To simulate a vehicle, we need a driver model. Its purpose is to follow a kinematic drive, e.q. NEDC or WLTC. ("NEDC.txt" in the example above). It generates signals that should be interpreted to be torque signal: accelerator (the above arrow exiting the driver), brake (the below arrow), combined (the midrange arrow); the latter is the only used in this experiment, and connected to power train in the figure above;

(2) Battery: It is a mathematical model of the EV electrochemical battery, to be substituted with the physical battery in our HIL experience;

(3) Eledrive: This simulates the electric drive (= inverter, motor). It contains evaluation of the driver efficiency, and of maximum and minimum torque and power and the various rotational speeds. Depending on the vehicle operation, it determines the power to be requested from the battery: this request is sent to the physical battery;

(4) Gears (= reduction gears), wheel and mass: they are self-explanatory;

(5) DragF (= drag force): It is the force that algebraically opposes to the vehicle movement, composed by rolling friction and aerodynamic drag;

(6) velSens (velocity sensor) is just a speed sensor: It gives actual speed information to the driver that consequently acts on its brake and accelerator to keep the actual speed as near as possible to the programmed kinematic cycle (the NEDC in figure). It pays the role of the dashboard on the physical vehicle.

The Modelica program implementing our electric vehicle according to Figure 2 is shown in Figure 4.

Here we have a modified version of electric drive (eledrive in the figure) model, which, starting from what is needed for the vehicle propulsion, computes the power requested to the battery.

Usually, the battery just delivers the requested power, and returns measures of voltage, current, and temperature, which are evaluated in the block "Battery data analysis." This block can compute and give as auxiliary outputs some battery parameters, e.g., internal resistance (based on an electrical model), internal temperature (based on a thermal model), etc. It also computes actual power P, which is sent to the electric drive to close the loop.

There might be cases in which the requested charge or discharge power is larger than allowable by the battery: in this case the battery voltage reaches some high or low threshold, respectively; in these cases, the battery control hardware in Figure 2, will adapt to these limits and reduce the exchanged power so that they can be satisfied, very much the same to what occurs in real-life vehicles. This happens for instance when the battery is nearly full and still requested to charge, or very low in charge and requested to discharge at high powers.

This behaviour of actual vehicle is reproduced in our system through the *Battery data analysis* block in Figure 4. This is one of the major advantages of this HIL system in comparison

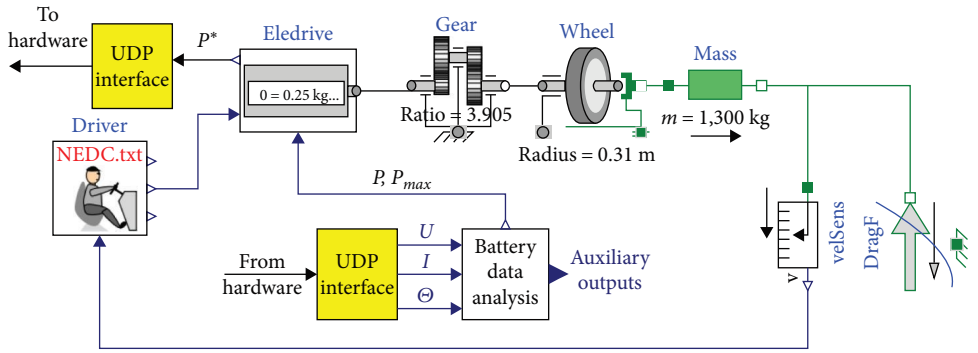

Figure 4: The Modelica diagram of the electric vehicle running in the vehicle simulator.

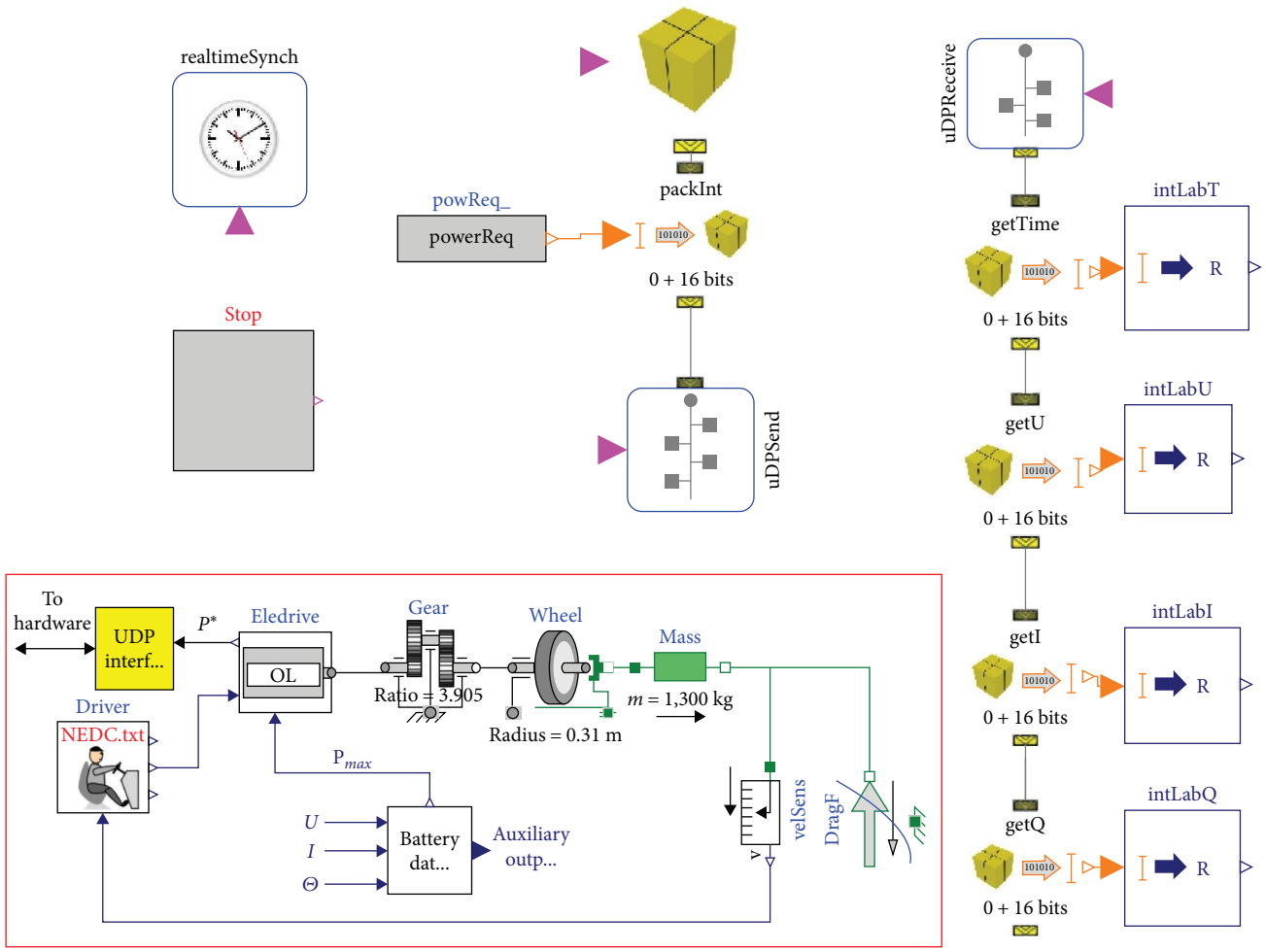

Figure 5: Use of Modelica DeviceDrivers library to exchange data via UDP.

with full simulations using a battery model according to the diagram in Figure 3: battery behaviour in such limit situations is very difficult to predict by means of battery mathematical modelling; the proposed HIL technique, instead, uses actual measured battery quantities to consider it.

The yellow boxes contain actual communication interfaces. They transfer data from and to the ethernet port, via UDP blocks using the DeviceDrivers Modelica library. In addition, in the case of the experiments done in this paper, they convert quantities from pack level to the cell level.

2.2.1. Interface with the External World. Communication is done through EDP packets. Library Modelica DeviceDrivers, allows to build packets from individual numbers, which ensures simultaneity of the exchanged values. Although, Modelica DeviceDrivers allows to exchange real numbers, in this experience,

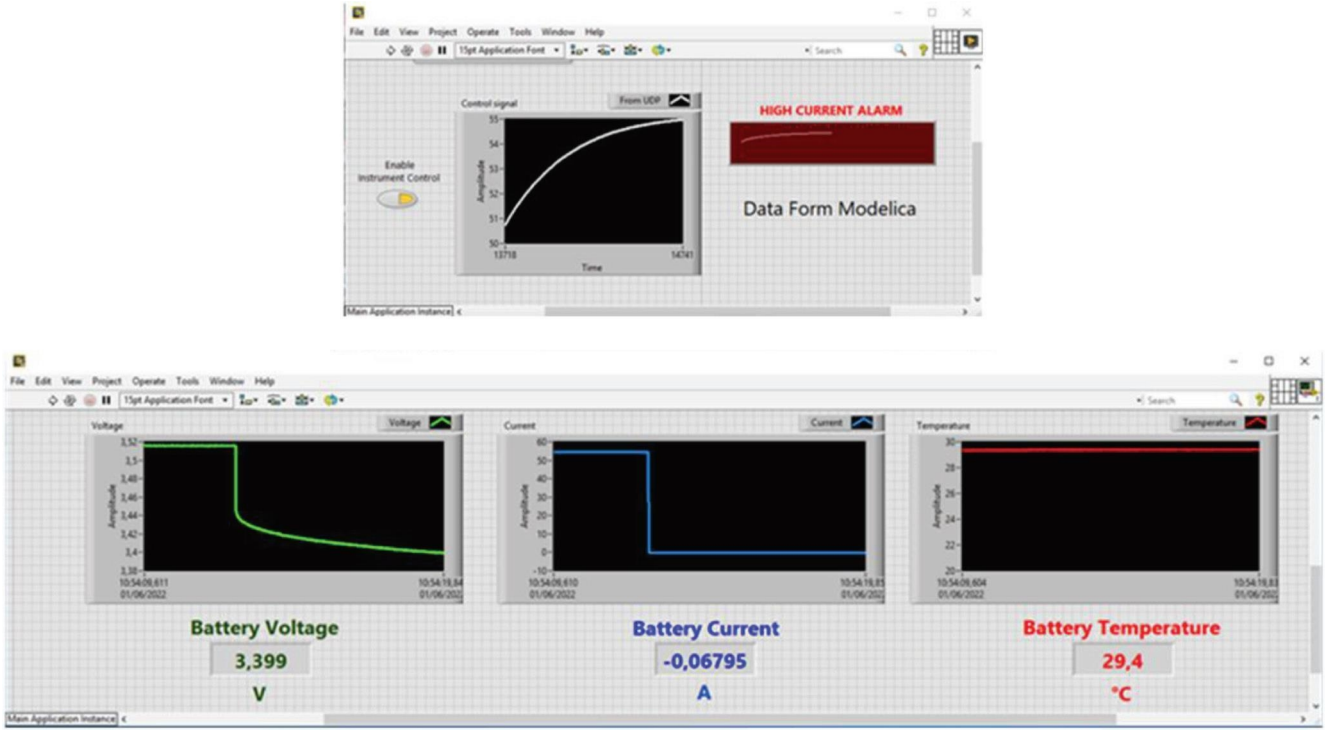

Figure 6: The three views of the LabVIEW program.

for simplicity, it was decided to send actual values through two-byte integer numbers.

From the Modelica part, therefore, data exchange were arranged as illustrated in Figure 5.

The simulator computes powerRequest (the value P^* shown in Figure 4), encodes it and sends it to the hardware. On the other hand, it retrieves from the hardware voltage, current, temperature, along with the time at which these samples were taken: these are the quantities from blocks getI, getI, getQ, and getTime, respectively. Before being sent, all values are converted in the defined two-byte format, and when received are decoded back. For instance, since in our experience we are sure to have currents within (−300–300A), the following formula was used, where subscript "i" indicates the integer-coded version of the current: $I = I_i/100 - 300$;

Here, some explanation of the new blocks:

(1) RealtimeSync: It is needed because Modelica is thought for simulations, and the simulator is designed to go as fast as possible. Here, instead, we must have the same speed as wall time, and therefore real-time synchronisation must be provided.

(2) STOP button: If for some reason we want to stop the experience, stopping the simulator leaves the power request at its last value, which could discharge of charge too much the battery. Obviously, some safety features are also implemented in the LabVIEW interface, but much better to first put powerRequest = 0, before actually ending the experience. This can be done using the STOP button, an instance of the free Modelica Library "UserInteraction," component "Inputs.trigButton".

(3) getTime, getU, getI, getQ, retrieve time, voltage, current, temperature as integer numbers from the UDP interface through the block uDPReceive.

(4) PackInt sends the power request to the UDP interface, thus the hardware, through the block uDPSend.

Note that the data received from the battery are combined in packets, each of which carrying time voltage, current, and temperature.

2.3. The LabVIEW Program. The software in the LabVIEW environment is organized into three distinct VIs operating in parallel, the front panels of which are shown in Figure 6.

The *RECEIVER VI* (upper plot in Figure 6) receives the parameters to be set (battery current I or power *P*) via UDP from the Modelica software and controls the instruments (power supply and electronic load) connected to the PC via GPIB (IEEE 488) interface to set the desired current (or power) charging or discharging on the battery. The *MEASUREMENT VI* (bottom of Figure 6) handles measurements, made by the acquisition modules (NI 9219) housed inside a NI cDAQ-9,172 chassis. The *SAFETY VI* (with blank front panel as operating without user interaction) handles safeties and alarms, and operates in parallel with the others and with independent measurements, allowing experiments to be automatically stopped in case of danger.

The following basic LabVIEW functions were used to communicate to Modelica:

(1) UDP Open function to open a UDP socket on a port (it returns a network connection *refnum* that uniquely

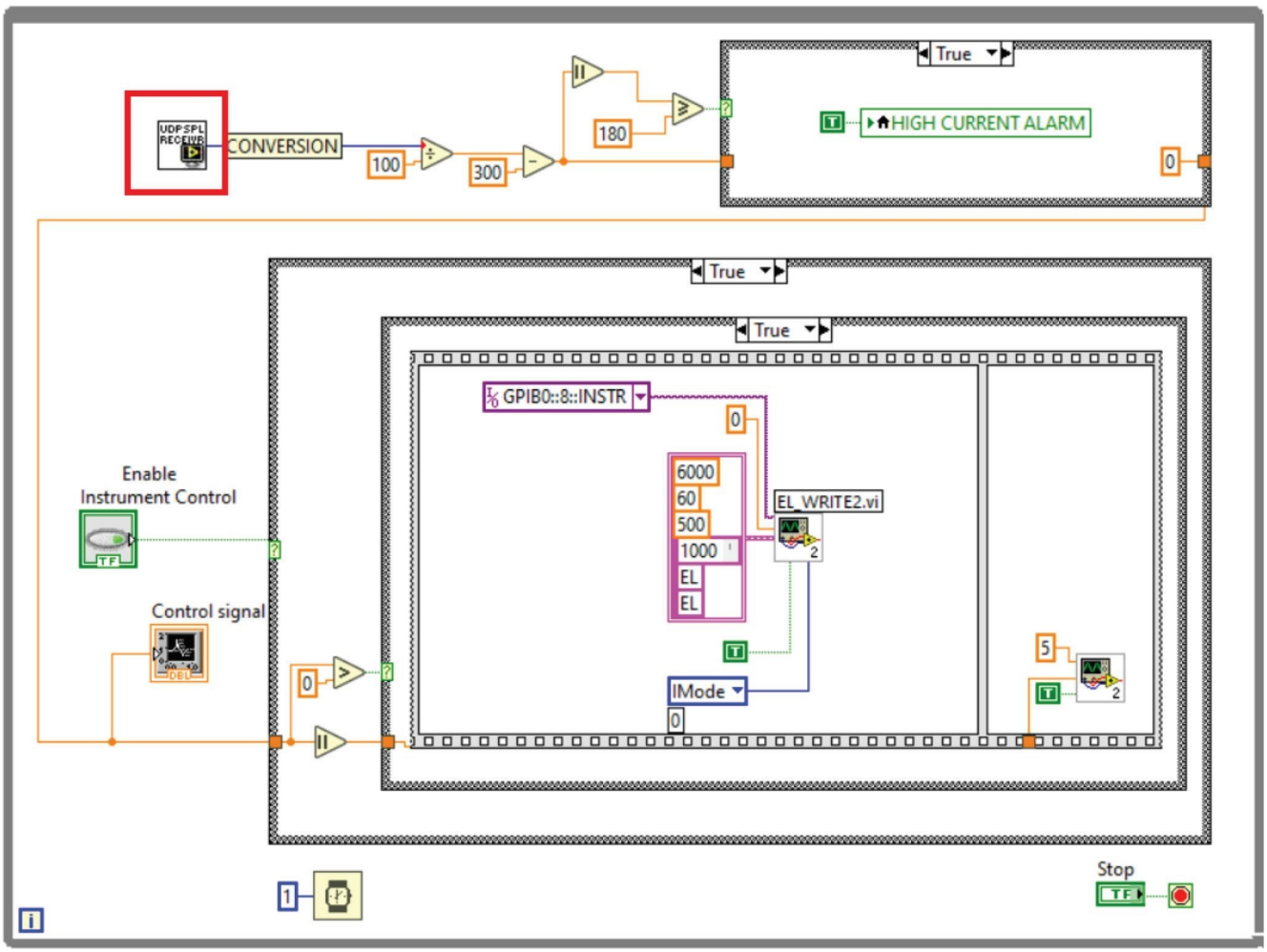

Figure 7: Block diagram of the LabVIEW receiver VI.

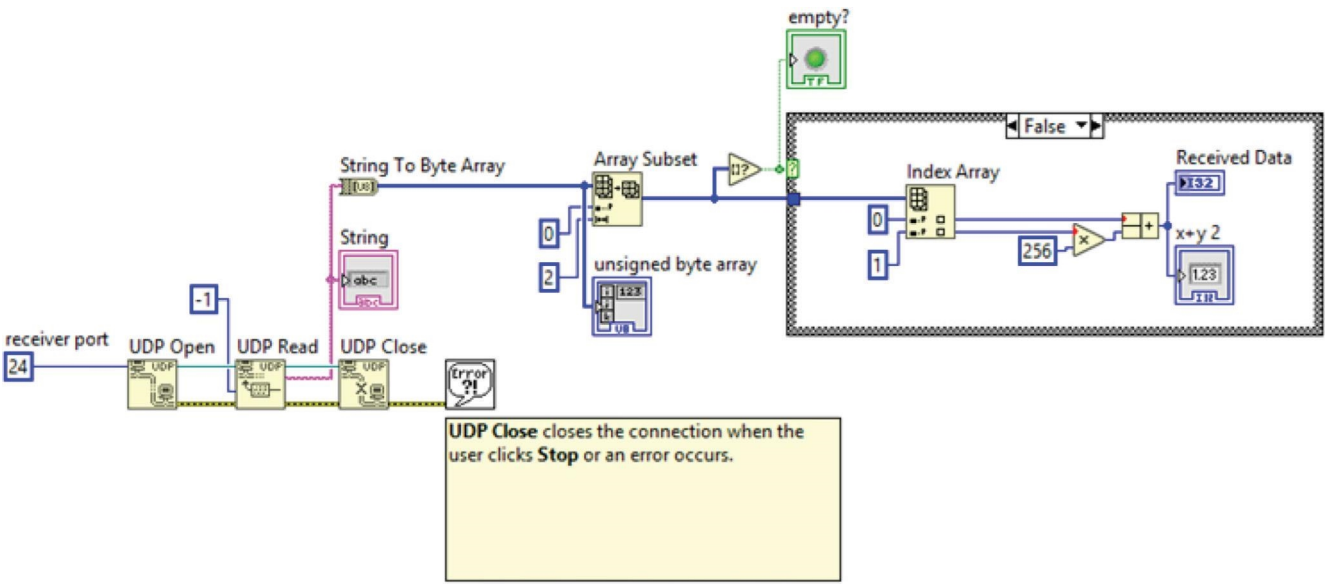

Figure 8: UDP communication LabVIEW SubVI.

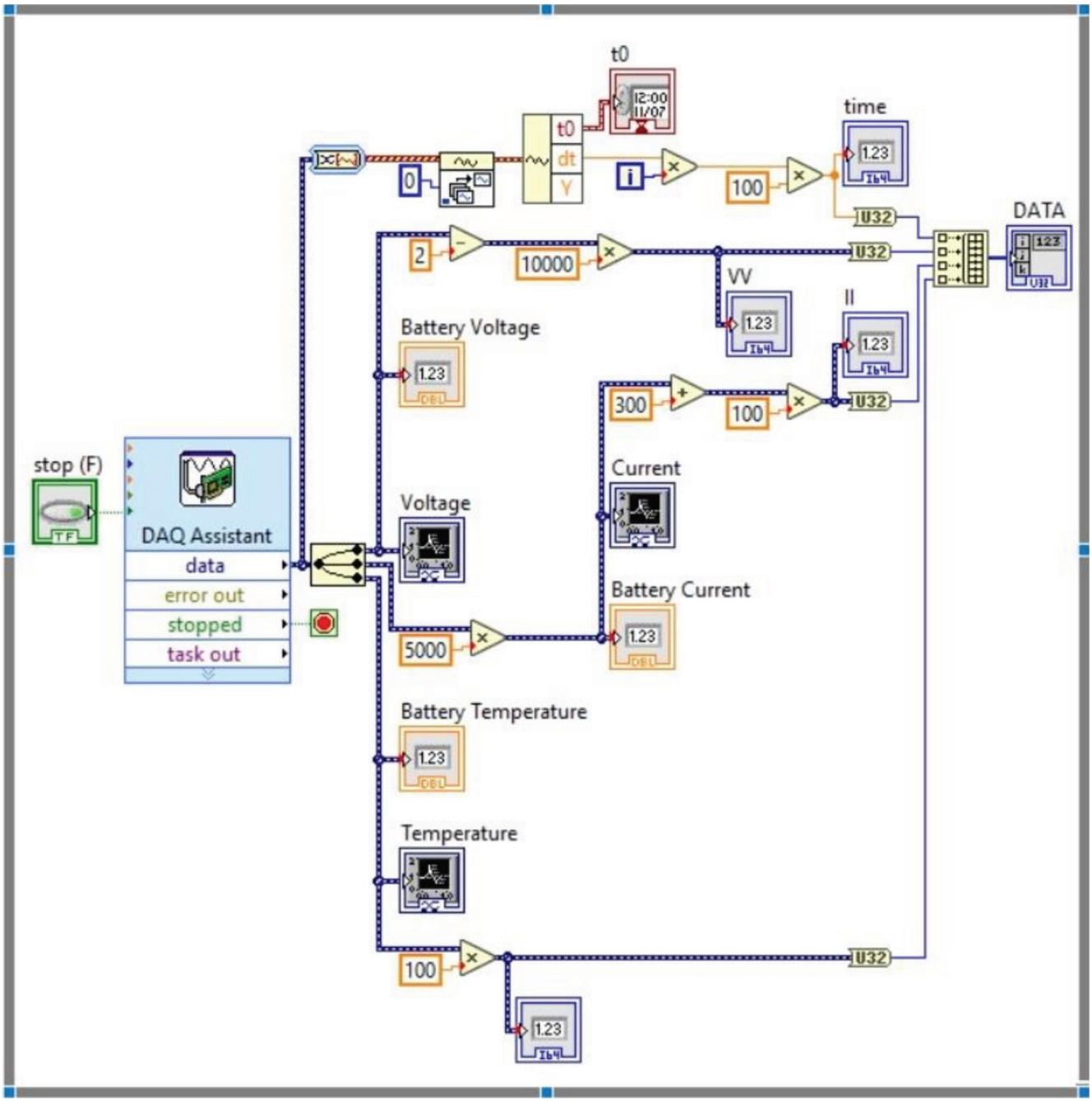

Figure 9: Measurement VI block diagram.

identifies the UDP socket and must be used to refer to this socket in subsequent VI calls);

(2) UDP Write function to send data to a destination;

(3) UDP Read function to read that data;

(4) UDP Close function at the end to free system resources.

The block diagram of the *RECEIVER VI* is shown in the Figure 7.

Data are read via UDP (subVI on the left inside the red square), converted (middle part) and used to set the battery charge or discharge current through the right part of the code that handles instrument control. As mentioned the first block on the left (red square) handles UDP communication and has the architecture shown in the following Figure 8.

The structure of the *MEASUREMENT VI* is shown in the following Figure 9. Measurement data are acquired using the DAQ Assistant block, then encoded and packaged to be sent via UDP to the Modelica software.

In particular, the appropriately encoded data are stored in a local variable (data block on the right in the figure), that

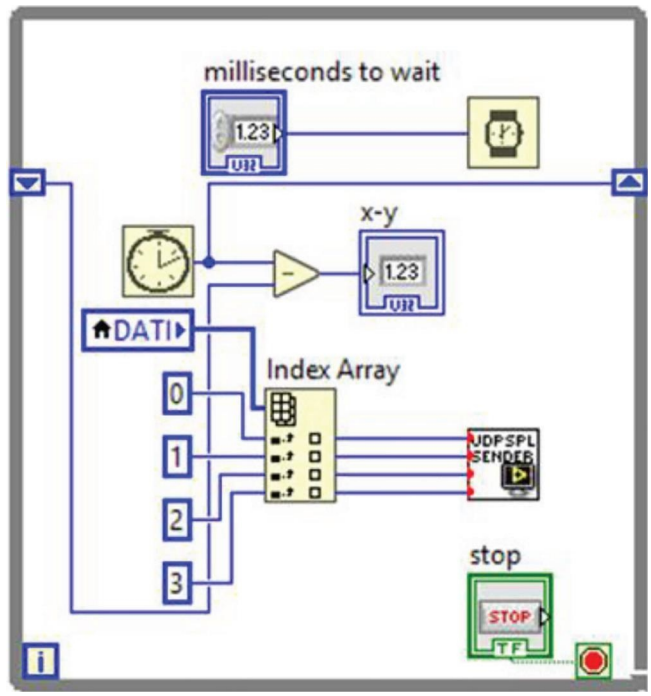

Figure 10: Data transmission to Modelica.

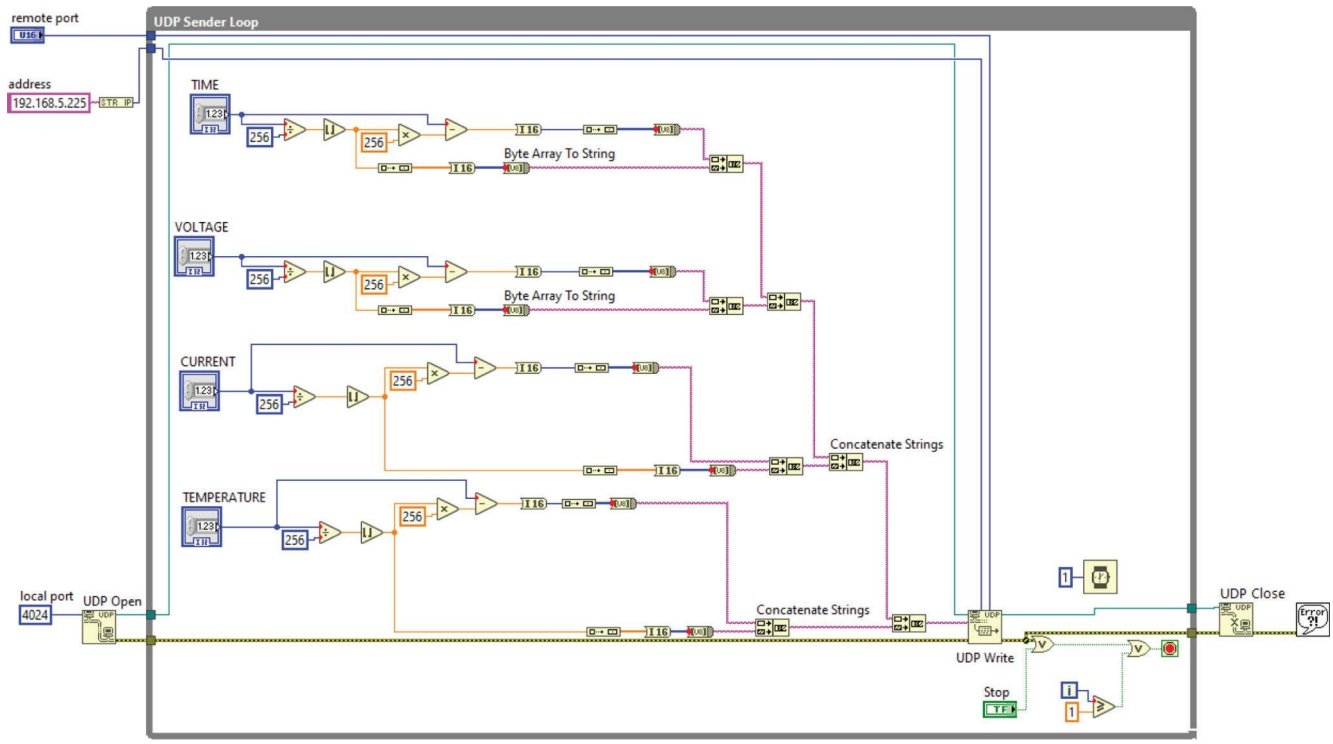

Figure 11: Data transmission block.

is read by a parallel while loop responsible for transmitting the data via UDP to Modelica. The block diagram of this part of the code is shown in the following Figure 10.

In our application data are transmitted to the destination frequently enough that a few lost segments of data are not problematic, so it was decided to implement direct data transmission without using the queuing mechanism.

Finally, the data transmission block, which uses the previously described functions (UDP Open, UDP Write, UDP Close), has the structure reported in Figure 11.

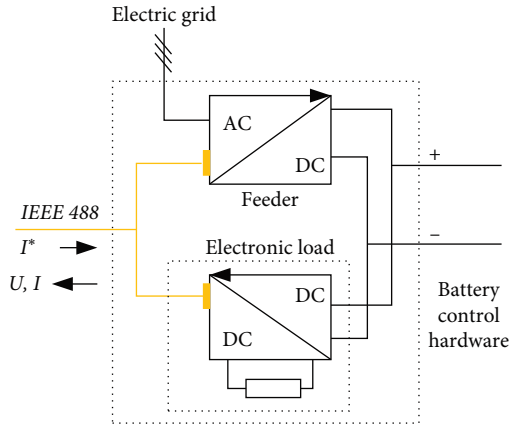

Figure 12: Battery control hardware used for the experiments.

2.4. The Battery Control Hardware. The battery control hardware must be able to apply to the battery both charge and discharge currents. This may be obtained with a bidirectional converter (typically a voltage source converter—VSC). However, this was not the case of the experience of this paper, where two separate pieces of hardware were available for charge and discharge (Figure 12).

3. Simulations (Virtual HIL)

To check the whole software environment, a "Virtual HIL" environment, has been created: In personal computer 1 (PC1) the software running contains the vehicle without battery (as per Figure 4) in personal computer 2 (PC2) the battery model which receives required current from PC1, through UDP interface, and sends back to the battery voltage, (actual) current, and temperature.

3.1. Simulation of a Simple Driving Cycle (Sort1). In this case, PC1 simulates a simple driving cycle, commonly used for busses, named Sort11 (It seems there are no scientific papers describing this cycle. Some useful info can however be taken from https://www.uitp.org/publications/uitp-sort-e-sort-brochures/).

The battery is simulated in PC2; the effect of splitting this simulation into two different PCs, is just demonstrative of the technology, since the battery actual behaviour on this example does not determine any special action to be undertaken in simulation running in PC1; the situation will be different in the next example.

Figure 13 contains a few curves of our vehicle running under Sort1; traction force and vehicle speed are from PC1, while battery voltage, although plotted from the program running on PC1, has been determined by PC2, from the current received through UDP communication from PC1. The battery current plot represents both I^* and I, since they are nearly indistinguishable from each other. Also the battery voltage is nearly indistiguishable from what could have been determined running the battery model inside PC1. Only when zooming very deep a small effect from the discretisation used is seen (figure bottom-right where voltage is slightly jagged).

3.2. Power Limitation due to Battery too High Voltage. The usage of actual hardware for mixed hardware-software experiments is important when the hardware influences the system's behaviour. In case of the battery, for instance, this happens when the battery becomes too full and cannot receive charge anymore, or too empty, and the vehicle performance degrades, up to a complete stop.

Before doing this kind of test with the system containing actual hardware, this has been simulated with the "Virtual HIL" arrangement shown in Figure 14.

The vehicle is simulated to go downhill for a very long time. At a given point, the battery is not able to receive any more energy, and regenerative braking must be stopped. In actual vehicles, mechanical, dissipative braking takes over; since in our simulation the mechanical braking addition is not simulated, we will see the vehicle taking excess speed in comparison with that requested by the driver.

The feedback from the battery behaviour on the power train control is as per Figure 15.

The generation of additional torque in data analysis is done here with just a simple PI controller without optimisation of proportional and integral gains, which is activated only when the cell's voltage overcomes the maximum allowed threshold.

Some results coming from the battery (simulated in PC2) are shown in Figure 16; if the voltage overcomes 3.95 V/cell, the electric drive is subject to an additional "accelerating" torque, that tends to reduce the regen braking, so that to avoid the battery to overcharge.

Although the control law is not optimised (it is just to show the concept) once the limit cell voltage, here set to 3.95 V, is reached, the additional torque has the effect to generate a current able to keep, with a limited overshoot, the voltage within the limit value. In the final part of the transient the system stabilises at a current 0, which corresponds to a battery having as OCV (open circuit voltage) the threshold value, i.e., 3.95 V.

Naturally, this reduction in braking torque from the battery in the vehicle will have to be compensated by addition of dissipative braking torque (mechanical brakes), not simulated here.

The results in Figure 16, refer to a battery having a rather larger variation of voltage during charge. Batteries with reduced variation like the one used in the experimental verification (next section) have different behaviour, but in principle the mechanism is the same.

4. Experimental Verification

4.1. First Interaction Tests. Before performing large tests, a lot of communication tests were carried out.

In Figure 17, we just show the graphical output of a current step required by a test Modelica program, and obtained through the arrangement in Figure 2.

We see the effects of a Modelica simulator requested current step, on LabVIEW VIs, controlling the actual hardware.

4.2. Power Limitation due to Battery too High Voltage with Actual Hardware. In the previous Section 4, the whole mechanism envisaged to make a Modelica simulation program interact with the hardware was simulated, using instead of the actual battery a second USB-interfaced personal computer (PC2).

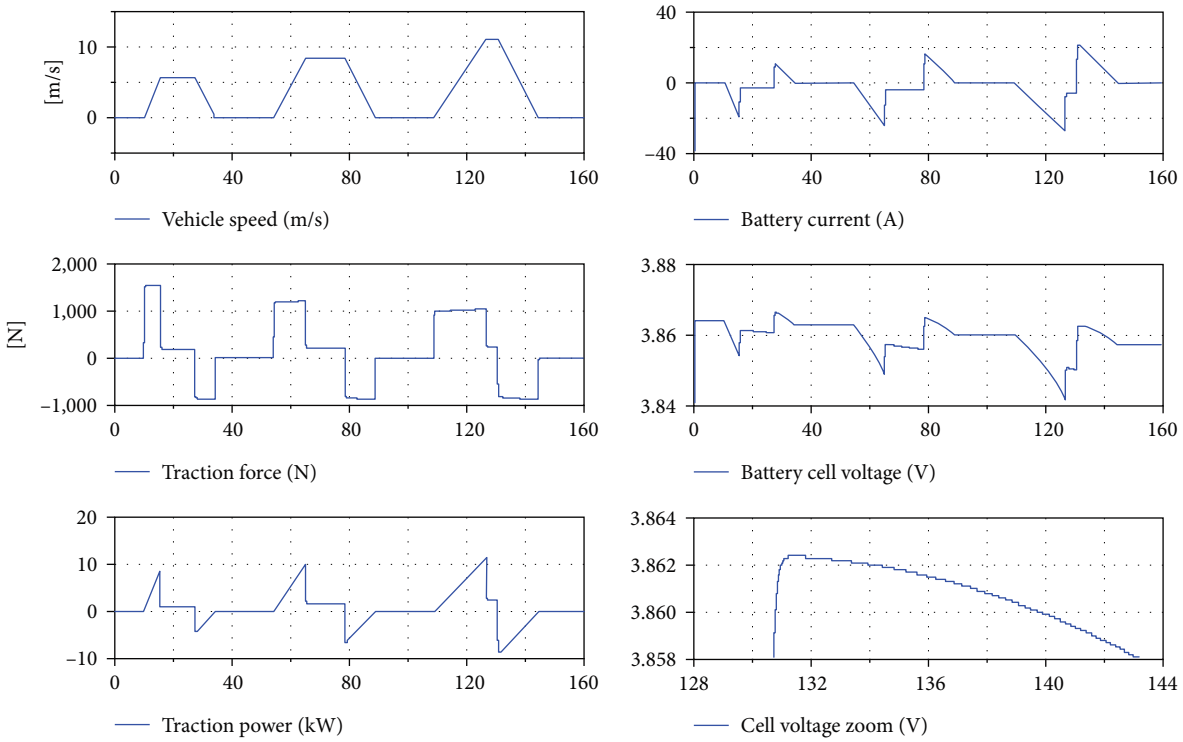

Figure 13: Results from the simulation of a vehicle running Sort1 cycle, under the setup of Figure 14.

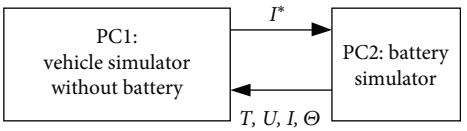

Figure 14: The "Virtual HIL" arrangement.

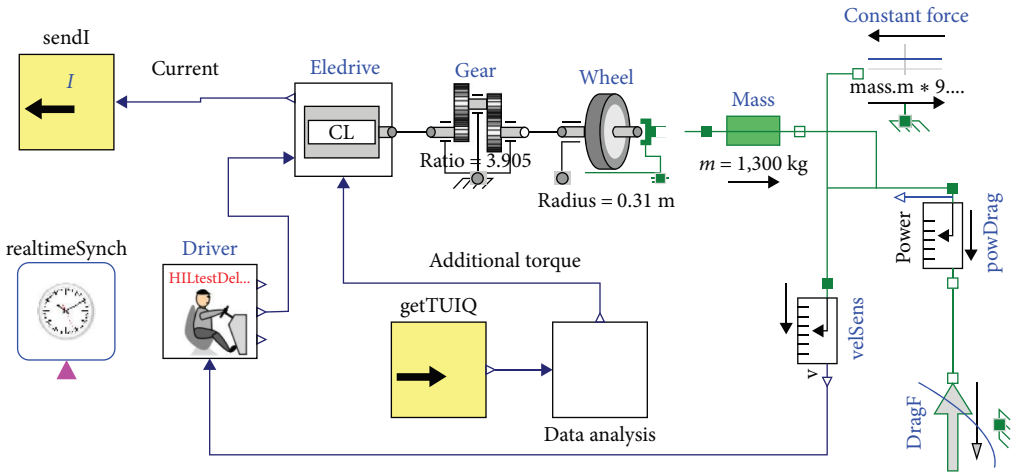

Figure 15: Diagram showing how power limitation is generated to avoid battery overcharge.

At this point, a final experimental verification with the full HIL arrangement as depicted in Figure 2 was made, for the evaluation of power limitation due to battery low voltage.

Unfortunately, we did not have at disposal the same cell whose model was used in the tests shown in Section 4 (a lithium–cobalt–oxide cell), but a different lithium cell, a lithium–iron–phosphate (LFP) cell which has different electrical behaviour, in particular a reduced variation of voltage ad different states of charge.

For the rest, the test was the same as in Section 4.2. The results obtained are shown in Figures 18 and 19.

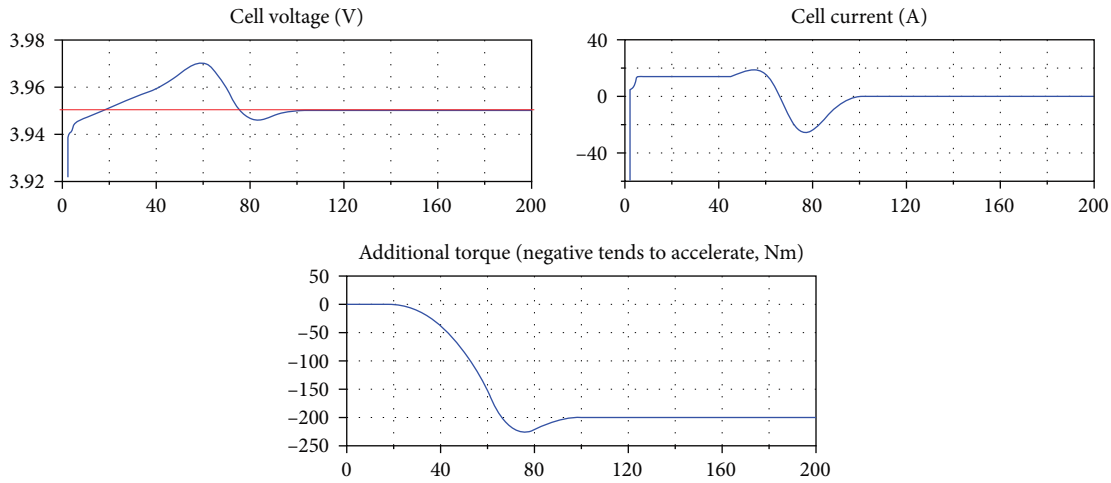

Figure 16: Plots showing the regen power limitation to avoid the battery to become overcharged.

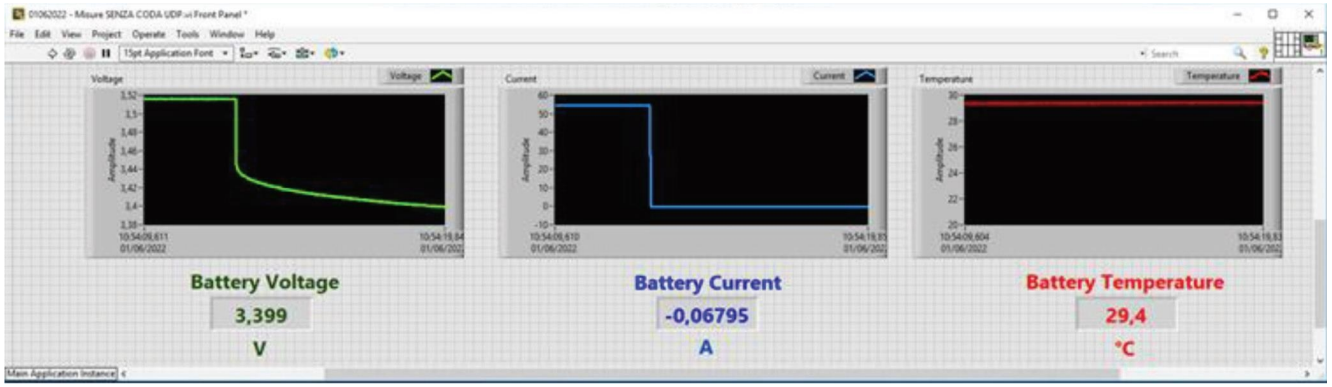

Figure 17: Interaction test with current step required by a test Modelica program.

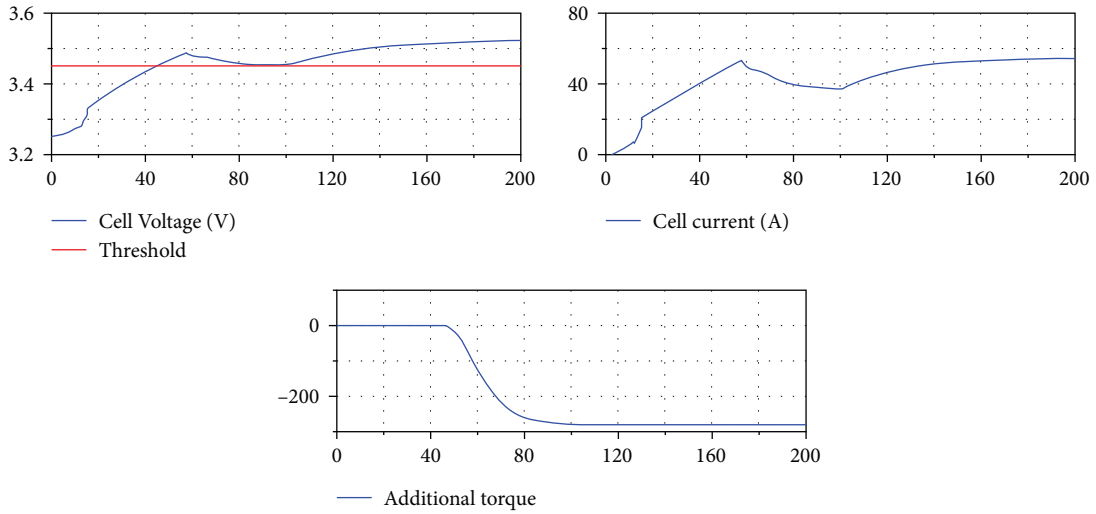

Figure 18: Results of the power limitation HIL test.

Consider Figure 18 which is the twin of Figure 16 obtained earlier; the following comments apply:

(1) Also in this case, when the voltage threshold is reached, the system starts to generate additional torque which reduces the regen braking (and induces subsequent mechanical braking);

(2) In this case at $t = 200$ s the current is still large (54 A), and the voltage continues to grow, but very slowly. It is apparent that in this experiment the integral part of

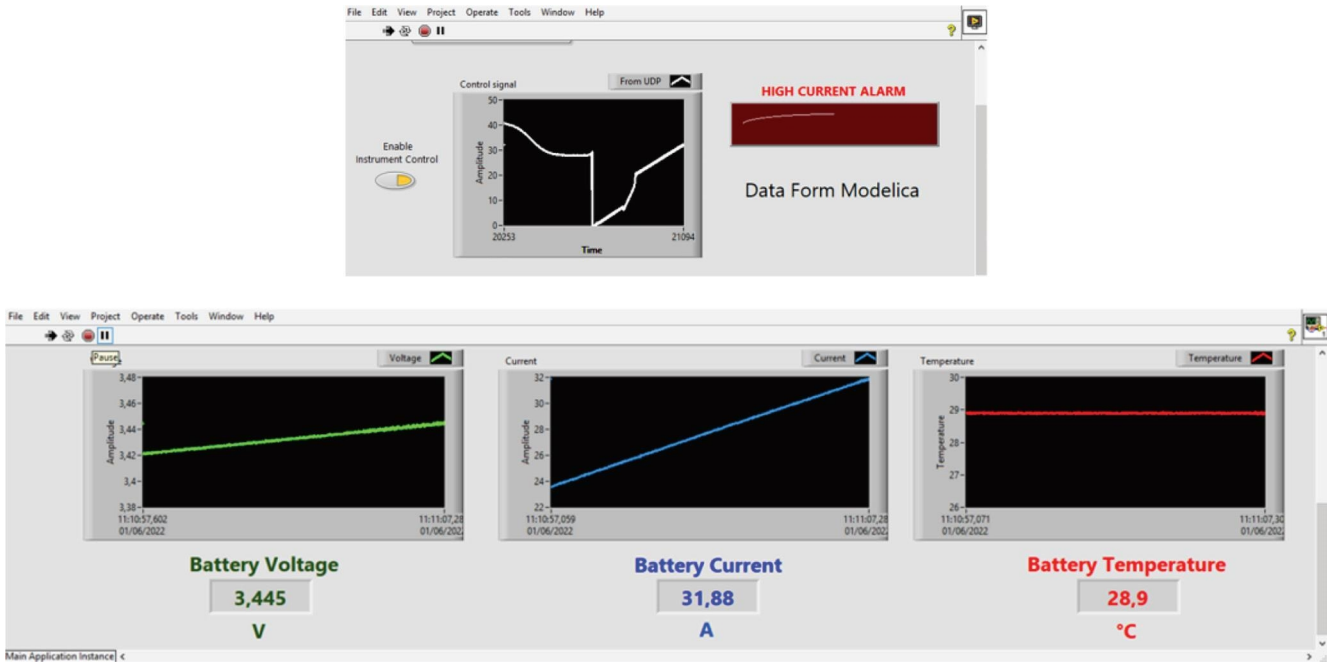

Figure 19: Some LabVIEW plots of the experience whose Modelica plots are in Figure 18.

the control takes nearly no effect, since the additional torque stabilises even when a finite-voltage error is present. To eliminate this error (and to obtain results like the ones in Figure 16), the integral part should be enlarged. This has not been done her, since the purpose of the experience was not to study the control to limit regen braking, but the effectiveness of the HIL;

(3) Once that this HIL architecture is created, experimental studies can be carried on the control of the vehicle in several situations, e. g., large positive and negative slopes, very large or small values of SOC, etc., that will be object of the further papers.

In the top upper part of Figure 19 we see the current requested by the Modelica-based vehicle simulator and corresponds to the plot shown as "cell current" in Figure 18 in the time window 50–110 s. In the lower part of the screen, we see VI monitoring cell parameters, with large degree of zoom.

5. Conclusions

This paper has shown that HIL simulations can be performed through Modelica-based software, communicating to LabVIEW-based control software and actual hardware. The considered case-study regarded an electric vehicle whose battery, instead of being software-simulated, is the actual battery, subject to simulation software current and voltage. For simplicity, here the actual battery was represented by a single physical cell; its current and voltage were scaled back to the battery level to be used in the vehicle simulator.

The obtained results are satisfying, given the inherent delays due to the UDP-based communication. It has been shown that to prepare these HIL experiences, a good procedure is to first create a "Virtual HIL" experiment, which facilitates fine-tuning the system, before going to the actual test with hardware.

Furter studies can consider converting the Modelica simulator into FMU, which is inserted in the LabVIEW program, thus dramatically reducing delays, and therefore further expanding the capability of the proposed technique.

Appendix

Research Data

The description of Sort1 Cycle in our Modelica code is made through a simple CSV file, which is composed by the following rows:

```
#1
#1st column: time, 2nd column: speed (km/hr)
double Cycle (13,2)
   00.00  0.00
   05.39  20.0
   17.22  20.0
   24.17  0.00
   44.17  0.00
   54.99  30.0
   68.37  30.0
   78.79  0.00
   98.79  0.00
  116.71  40.0
  120.60  40.0
  134.49  0.00
  150.00  0.00
```

This file allows reproducing the left part of Figure 13. The right part, however, depends on the specific battery tested.

Data Availability

The paper uses a tiny amount of data, which is directly included in the section "Research data" of the paper.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

We want to warmly thank Roberto Di Rienzo for his extremely valuable help on setting up UDP communication.

References

[1] M. Tiller, *Introduction to Physical Modeling with Modelica*, Springer International Series in Engineering and Computer Science, Kluver Academic Publishers, 2004.
[2] P. Fritzson, *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*, Wiley, 2014.
[3] S. Barsali, P. Bolognesi, M. Ceraolo, M. Funaioli, and G. Lutzemberger, "Cyber–physical modelling of railroad vehicle systems using Modelica simulation language," Ajaccio, Corsica, Railways, 2014.
[4] M. Ceraolo and G. Lutzemberger, "Electro-mechanical modelling and simulation of railroad vehicle systems using Modelica language," *International Journal of Railway Technology*, vol. 4, no. 3, pp. 37–67, 2015.
[5] M. Ceraolo, "A new Modelica electric and hybrid power trains library," in *Proceedings of the 11th International Modelica Conference*, pp. 785–794, LiU Electronic Press, Versailles, France, September 2015.
[6] "Dymola systems engineering modelling and simulation platform," https://www.3ds.com/products-services/catia/products/dymola/.
[7] "OpenModelica official internet site," https://www.openmodelica.org.
[8] B. Thiele, T. Beutlich, V. Waurich, M. Sjölund, and T. Bellmann, "Towards a standard-conform, platform-generic and feature-rich Modelica device drivers library," in *Proceedings of the 12th International Modelica Conference*, pp. 713–723, LiU Electronic Press, Prague, Czech Republic, May 2017.
[9] https://github.com/modelica-3rdparty/Modelica_DeviceDrivers.
[10] J. Postel, *User Datagram Protocol, Internet Engineering Task Force*, Association for Computing Machinery, 1980.
[11] D. E. Corner, *Internetworking with TCP/IP—Principles, Protocols and Architecture*, Pearson, 6th edition, 2001.
[12] International standard ISO 11989-1, "Controller area network (CAN)," 2015.
[13] https://www.ni.com/it-it/shop/labview.html.
[14] "NEDC cyle is defined in regulation UNECE (United Nations Economic Commission for Europe) N. 101, and its updates".
[15] M. Tutuianu, P. Bonnel, B. Ciuffo et al., "Development of the world-wide harmonized light duty test cycle (WLTC) and a possible pathway for its introduction in the European legislation," *Transportation Research Part D: Transport and Environment*, vol. 40, pp. 61–75, 2015.