

Multi-Disciplinary Design Support using Hardware-in-the-Loop Simulation^{*})

Peter M. Visser, Marcel A. Groothuis and Jan F. Broenink
 Twente Embedded Systems Initiative,
 Cornelis J. Drebber Institute for Mechatronics and Control Engineering,
 Dept. of Electrical Engineering, University of Twente,
 P.O.Box 217, NL-7500 AE Enschede, The Netherlands
 Phone: +31 53 489 2788 Fax: +31 53 489 2223
 E-mail: p.m.visser@utwente.nl

Abstract – This paper describes a method using Hardware-in-the-Loop Simulation as a means for multi-disciplinary design support. The method presented here, aims at supporting the design of heterogeneous embedded control systems. The method considers the implementation process as a stepwise refinement from physical system models and control laws to efficient control computer code, and that all phases are verified by simulation. Since many systems are distributed in nature we use a link driver library based on the CSP channel concept to support this. Communication peculiarities are encapsulated by these link drivers. Experiments with a basic mechatronic set up show that the HIL Simulated system behaves as the real system and can be successfully incorporated.

Keywords – Embedded control systems, real-time, model-based approach, Hardware-in-the-Loop Simulation

I. INTRODUCTION

Design trade-offs are often made in a single discipline, e.g. mechanical engineering, without regarding the system as a whole. This can result in a *not*-optimal system design and possible integration problems. A model-based approach and hardware-in-the-loop simulation are recommended as a means of multi-disciplinary analysis.

Design of complex heterogeneous systems requires an integral approach for a systematic architectural design, modeling, analysis, and validation methodology. An example is a document printing system. Such systems usually comprise several of the following disciplines: mechanics, electronics, software and chemistry.

In this paper, a case study is presented to demonstrate design support using hardware-in-the-loop simulation. The design trajectory as proposed in [1] will be used as a starting point. An attempt is made to illustrate the boundaries between the involved disciplines, hence to reflect how mono-disciplinary design changes can influence the system as a whole.

First, a brief introduction is given of Hardware-In-the-Loop Simulation (HIL Simulation or HILS), a kind of real-time simulation [2]. Then, HIL Simulation is brought under the context of a structured approach to embedded control systems software implementation as presented in [1]. A benefit in this case is that it allows for concurrent engineering in an early stage.

Section four describes the HILS platform and the tools that support the proposed method. Various choices, such as using FPGAs as I/O devices, to build the HILS platform are elaborated on.

A demonstration setup shows a sample configuration and its results.

II. HARDWARE-IN-THE-LOOP SIMULATION

Testing and simulation of control algorithms is an important phase in the development of embedded control systems (ECSs). Different types of simulation are possible during the design process of a controller, ranging from simulation without time limitations, to partial real-time simulation in which only some parts of the complete control loop are simulated (see Figure 1). Real-time simulation means here that the simulation is performed such that the input and output signals show the same time-dependent values as the real component [2].

In this research, Hardware-In-the-Loop simulation (HILS) involves connecting the actual ECS to a computing unit with a real-time simulation model of the plant (simulation of the hardware in the control loop, which is the middle situation of Figure 1. The architecture of the actual experimental setup is shown in Figure 2.

^{*}) This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program

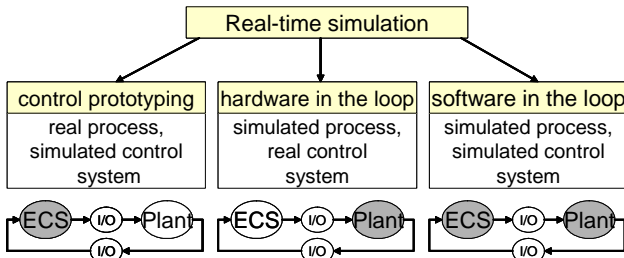


Figure 1: Real-time simulation methods (from [2]).

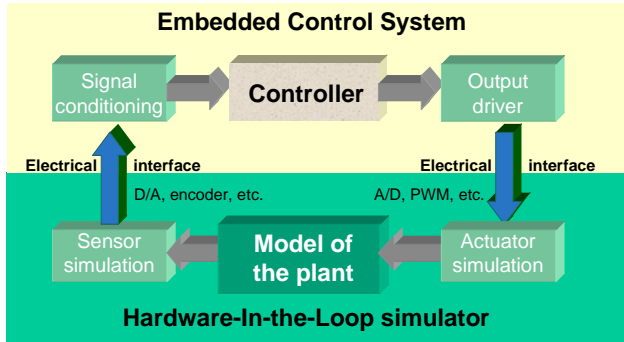


Figure 2: Used HILS set up.

Compared to ‘ordinary’ simulation, extra computer hardware is needed: the simulation has to run in real time, the I/O interfaces have to be available. Furthermore, conversion from the I/O-signals back to computer numbers must be constructed (the blocks “Sensor simulation” and “Actuator simulation” in Figure 2). These interfaces are not commonly available, and thus might be costly to produce.

The main advantages of HIL Simulation are:

- Plant models used during off-line design and simulation for the controller development can now be used for the ECS testing. This implies that, at software testing, the stubs representing the plant now can be proper models instead of simple signal generators. This results in better quality of the ECS tests, allowing for a less complicated integration phase.
- Software design and testing can be moved to an earlier design phase, i.e. before a first prototype is available, allowing concurrent engineering between the different design disciplines. This results in a shorter time to market.
- The ECS software changes can straightforwardly be checked for consistency with the design. Test software and test data written in the control design stage can be reused easily.

An additional benefit is that plant models used for off-line design and simulation during the control development can be reused for the ECS testing.

III. DESIGN METHOD

The purpose of the design method proposed in this section is to bring disciplines together in an early stage. This is to avoid the pitfall of complex system integration in a later stage when the disciplines have no, or not enough, interaction in earlier stages. A design trajectory subject to this pitfall is the following:

First the mechanical engineers make a mechanical setup, or prototype. Then electrical engineers add electronic interfaces. In a final, stage it is the task for software engineers to “glue” everything together as one working system. When problems occur the whole cycle may be executed once more, which is costly.

The design trajectory proposed in [1] is used, see Figure 3, and will be used in combination with hardware-in-the-loop simulation.

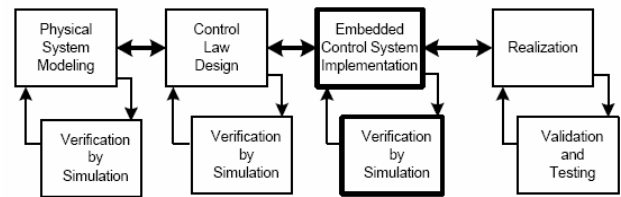


Figure 3: Design trajectory for embedded control systems

The trajectory is summarized into the following steps:

- Model the plant and design a controller; verify them by simulation.
- ECS implementation; verify by simulation.
- Realization: real prototype or plant with real ECS: validate, measure and test.

By using hardware-in-the-loop simulation, this trajectory is *not* followed as a waterfall approach (followed from the left to the right) but iterated in a micro-cycle fashion [3]. The cycle fashion of this trajectory is elaborated on.

First a simple physical model is made, which is based on basic physical principles. A corresponding simple control law is designed. Both controller and plant will be simulated, Figure 4 depicts this.

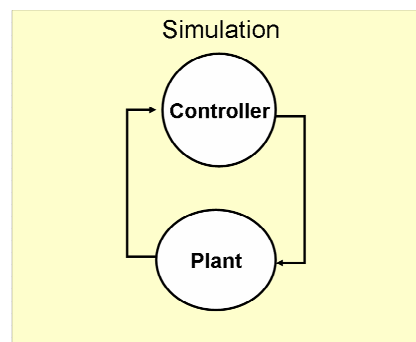


Figure 4: Simulation of controller and plant

The disciplines involved in this step are control engineering and mechanical engineering. Trade-offs can be made between controller complexity and plant dynamics.

Following this step, a simple ECS implementation is made. This will be run on the hardware-in-the-loop simulation and evaluated. Figure 5 depicts this.

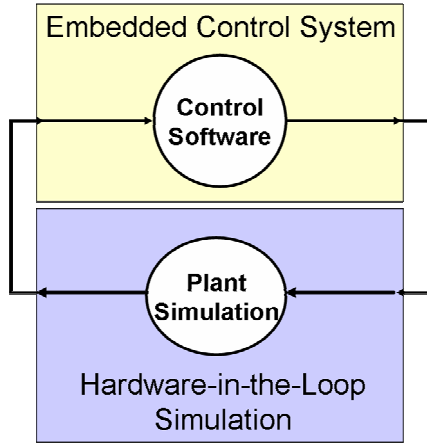


Figure 5: ECS and HILS experiment

The disciplines involved in this step are control engineering, mechanical engineering and software engineering. The system should behave in exact the same way as in the previous step. A common error made is in translating the control law to computer code.

Hardware-in-the-loop simulation “forces” the disciplines to cooperate on the same experiments. This allows cooperation and a better focus on the system as a whole.

In this step trade-offs, system design choices, can be made over the boundaries of a single discipline. As an example the control software consumes 90% of the system resources. This could lead to one of the following choices: a faster hardware platform, a simplified control law, change the plant which allows a simplified control law.

The plant and controller can be altered, either to reflect a design choice or to add details, and both steps can be cycled another time. Such a cycle can be performed in hours/days with the HILS. Instead when using a prototype, changes can be made much slower: weeks/months. Another benefit is that *Concurrent Engineering* can be used in this stage.

The next step is to take effects due to non-idealness of computer hardware [1] into account. The electrical engineering discipline is needed. First the effects will be added in the simulation domain, see Figure 6. When satisfied, the next step is to perform the hardware in the loop simulation, see Figure 7.

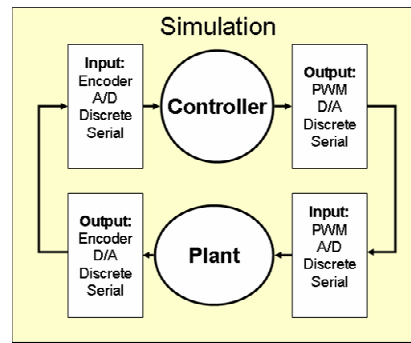


Figure 6: Simulation of controller and plant with I/O

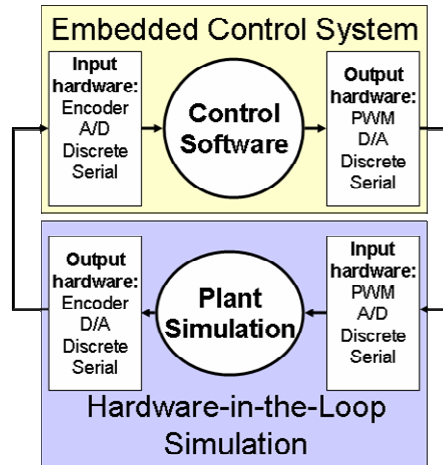


Figure 7: ECS and HILS with I/O experiment

After testing and evaluating extensively, reaching a stable ECS and HILS, a prototype will be build. The HILS will be replaced and the ECS can be connected to the plant without any changes, see Figure 8.

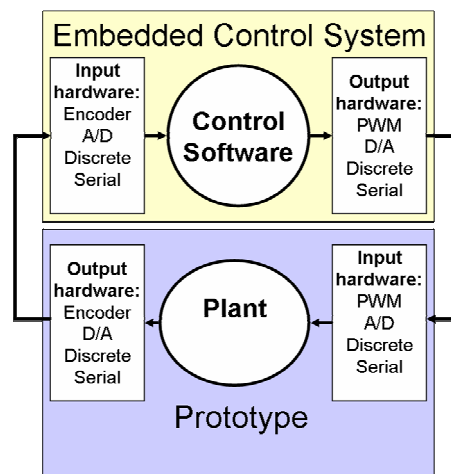


Figure 8: ECS and Prototype

In the time the prototype is being build additional features can be added to the ECS e.g. error handling.

In case of a large complex plant more than one ECS can be controlling the plant. The method in this case is the same but there will be more focus on the software architecture. The HILS must support multiple ECSs and the next section show how this can be handled.

IV. PLATFORM AND TOOLS

This section describes the hardware and software of the platform and tools used for the HILS. First the hardware platform of the ECS and HILS is discussed. This is followed by the choice for the operating system. Then the interaction between the ECS and HILS, via I/O, is important and will be elaborated on. Finally the code generation process will be discussed.

The platform must be suited to perform distributed control hence more than one ECS target is wanted. The multiple ECS targets need to communicate, the CAN fieldbus is chosen for this purpose. The ECS setup is chosen to be a set of four embedded X86 PC/104 processor boards, connected to each other by a CAN bus. The choice for PC/104 PC's and not just normal pc's equipped with a CAN card, was made because it is an industrial standard for small embedded pc's. Use of the CT library [4] is a second important reason for X86 compatible PC's.

The HIL simulation setup is a Proof of Concept setup. It will be used to test controller architectures and parameter changes in the plant models. To prevent running into problems because of the PC/104 hardware limitations, the choice is made for a powerful processor and for enough memory. This enables a broader range of experiments. For (re)configuration of the ECS, during testing, the ECS pc's should be equipped with a network link connected to a central control and configuration PC.

To run the simulation models for Hardware-In-the-Loop simulation, one (or more) powerful processing units that are able to simulate the dynamics of plants in real-time (calculations and I/O for generation of new output signals should be finished before the next sample moment).

The final hardware choice for the ECS is a PC104+ CPU board with a 600 MHz CPU, supplied with 256 MB RAM and a 32 MB Flashdisk with CAN fieldbus support. The HILS is chosen to be a fast 3GHZ x86 compatible pc. The architecture of the setup is depicted in Figure 9.

Now that the hardware architecture has been defined the accompanying operating system and software requirements will be discussed. To boot the ECS PC's an operating system is required.

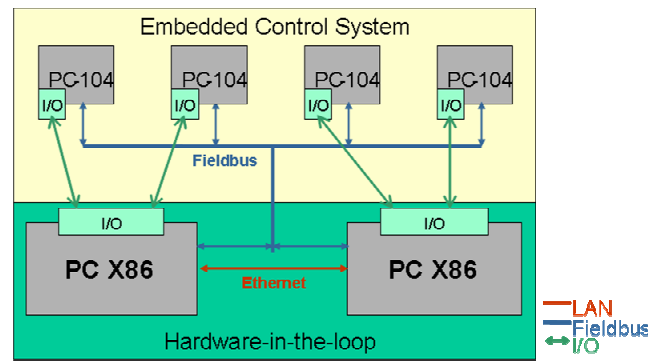


Figure 9: Architecture of the setup

The operating system should be able to run CTC(++) programs and 20-SIM models (via code-generation). An operating system with *hard* real-time capabilities is required for the ECS setup. Both DOS and (RT)Linux are in use for this purpose. DOS is a simple 16-bit operating system without multitasking capabilities and can therefore be considered as real-time capable. Linux itself is not capable of guaranteeing the predictable time limits needed for real-time tasks. The Linux kernel is designed to optimise *average* performance. This means that all processes will get a fair share of CPU time. For real-time programs, precise timing and predictable timing is more important than average performance. Therefore, Linux needs some real-time extensions like RTlinux or RTAI, which combines a hard-real-time kernel (e.g. Adeos or RTHAL) with the normal Linux kernel. See for more information [5]. The choice of operating system was made in favour of Linux together with RTAI. This is mainly due to the limited driver availability of DOS and the lack of network support for remote configuration purposes. Furthermore the Linux installation for the ECS uses a non-standard C library, called uClibc [6], which is designed to reduce the size of the standard Glibc library on embedded targets. The embedded Linux installation has no on-board compilers because of their size.

The CT library is a useful tool for the development of controller software in a distributed environment. It offers a simple method to divide processes over several processors by means of replacing internal channels by remote channels [7]. The library disconnects the development of the software and the addressing of hardware and external communication links.

An important part of the setup is the I/O interface between the ECS and the HILS. The requirements of the I/O interface depend on the connection with the HIL simulator and on the exact boundary between simulation and real hardware. Figure 2 shows four green blocks that represent the I/O devices between the controller/ECS and plant/HILS. The I/O configuration must be able to comply with Figure 5 and Figure 7.

A brief introduction into the basics of input and output are given from the perspective of a (real-time) computer system. After that, the choice for the I/O devices is given.

Every I/O signal has two dimensions, its value and the time. This time can have different roles in a real-time system [8]:

Time as data: time of a value change is important of later analysis of the consequences of the event, e.g. how long did an event endure.

Time as control: a value change may require immediate action of the computer system, e.g. for an airbag sensor.

In the context of computer hardware, the value is the contents of a register and the time is the trigger moment at which the register contents are transferred to another subsystem. The role of the time for a particular I/O signal, gives different requirements for the hardware and software. Delay is critical in a time-as-control situation. These kind of signals require minimal delays and must be treated in hard real-time. On the other hand, in the time-as-data situation, a global time base is more important than a minimal delay. Some delay is no problem, as long as the delay is known. E.g. a control action that should occur precisely at the intended moment: the next sample time.

Different ways exist for observing the state of an input value: sampling, polling and interrupts. Observing the value on equidistant times is called sampling. The time interval between two samples is constant. Polling and sampling are almost the same. With sampling, the sensor keeps the last known value until the computer reads its value and with polling, the computer plays the role of memory. With polling, the computer reads only the most recent value and events that occur between two sampling point are not 'seen'. Polling is enough if only the most recent value is of interest. If every event (state change) is important, one should use sampling.

Using interrupts, the change in the status of an input value can interrupt the running software and start an interrupt service routine to service the event. Kopetz [9] states that the interrupt mechanism is therefore dangerous, because it interrupts the running (controller) software, introducing an extra delay. Interrupts are only needed for external events that require a shorter reaction time than possible with regular sampling (e.g. for time-as-control signals).

The types of the sensors and actuators are important for the choice of the I/O. It is not just analog or digital I/O that is required.

The most common types of sensor/actuator I/O signals are analog I/O, digital I/O and intelligent instrumentation. Examples of analog I/O are current, voltage and frequency. Examples of digital I/O are state (on/off),

value (multiple bits) and time encoded signals (pulse width modulation, encoder position). Intelligent instrumentation are sensors with build-in intelligence. The real sensor/actuator interface is hidden behind a build-in microcontroller. Some of these sensors/actuators can be connected directly to a standard I/O interface or field bus.

The manner in which a computer system approaches the I/O is through one of the following three methods: programmed I/O, memory-mapped I/O, direct memory access (DMA). Interrupt driven I/O access can be seen as the fourth method for I/O access. However, it is an implementation of one of the above methods in which an interrupt signals the completion of an I/O transfer or a request for an I/O transfer.

With programmed I/O, the CPU is involved in the transfer of I/O data. These CPU efforts cost time (approximately 1 μ s per inb/outb call) which must be taken into account for the real-time performance.

Memory mapped I/O provides access to I/O data via designated memory locations that work as virtual input/output ports. Memory-mapped I/O is faster than programmed I/O, if the response time of the corresponding I/O device is similar to the response time of regular memory.

Direct memory access gives I/O devices access to the memory without CPU intervention, resulting in a faster data transfer from and to the memory. The disadvantage of DMA is that the CPU cannot perform a data transfer during DMA. The CPU has to wait (or proceed with calculations that do not require external data) until the DMA controller allows the CPU data transfer. This influences the real-time behavior when a lot of data is transferred using DMA.

The HIL simulation in context of the systems under scope does not require large data transfers, so DMA is not required.

The I/O device chosen to comply with the given description of I/O is an FPGA, because all these hardware I/O devices can be emulated by software configuration in the FPGAs. The tremendous advantage of FPGAs over real I/O devices is their high configurability. In context of HILs it is a prerequisite that the software of the ECS can remain the same and only the configuration of the FPGAs need to be altered. The FPGAs in the setup are chosen to be fast enough in the field of mechatronic control applications. The FPGAs are on a PCB board which is available in both a PCI and a PC104-PCI bus. The FPGA has 200.000 system gates and 56k blockram and can run up to 200MHz. The PCB board, anything I/O board, has 72 general purpose digital I/O pins [10]. All the four green blocks in Figure 2 can be realized with the same FPGA.

For efficient use of the HIL simulator and the ECS, a comfortable experimental environment is required. For this purpose 20-sim [11], which has the capability of automated code generation, will be used. Figure 10 depicts this environment. Relevant signals can be visualized for analysis in 20-sim using recorded data. An extension is on-line data logging via a network connection during a HI simulation experiment, this is not yet implemented. Furthermore the feature of changing parameters on-line is not yet implemented.

V. DEMONSTRATION SETUP

A setup, called *Linux*, is used as mechatronic platform. Linux consists of one motor and one encoder which are both on the same axis. The motor drives a wheel which is connected by a rubber band to another wheel, the load. Both the model of Linux and a controller are designed in 20-SIM [11]. The automatic code generation for this example is depicted in Figure 10.

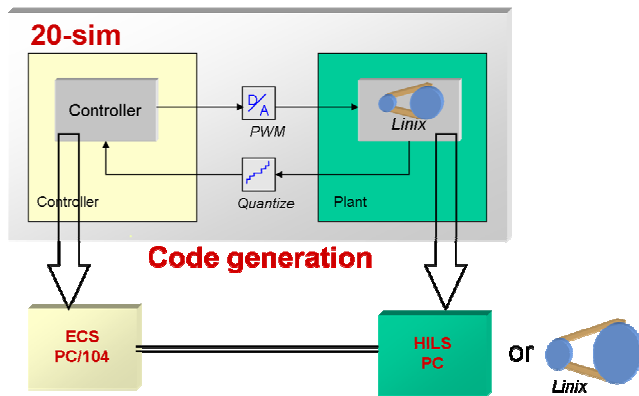


Figure 10: Code generation for HILS

As proof of concept the simulation results of the model are compared with the real-plant and the HILS.

A. The plant model

The model of the plant is depicted in Figure 11 below:

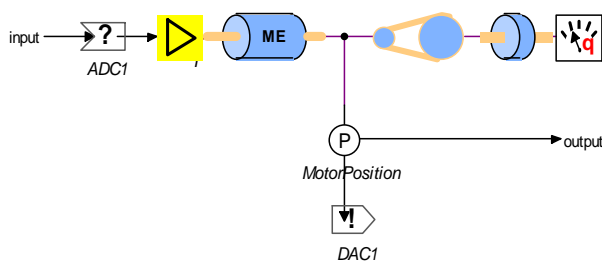


Figure 11: The Linux plant model

The plant was simulated as a separate entity before designing the controller conform the design trajectory. The \square and \square special blocks reflect the I/O, which is necessary for code generation.

B. The Controller

The controller for the Linux setup is a discrete PID controller. Figure 12 shows the internals of this controller. The controller inputs are the reference position and a feedback of the real position. The output is the motor steering signal which will be converted into a corresponding PWM duty cycle.

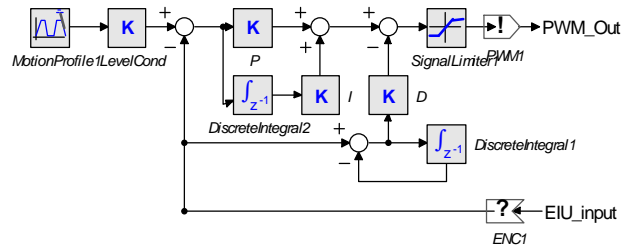


Figure 12: The discrete PID controller of Linux

C. Simulation and results

All state-variables and other signals both of the plant and the controller can be inspected in the simulation environment. The most interesting signals are the angular positions of the motor and the load, hence the signals from the position sensors in the model. On the HIL simulator only the *input* and *output* were monitored, in order to compare the results with the real plant (on the real plant, only the inputs and outputs can be captured). Additional signals can be inspected but have to be added, by special blocks, in the model. The same holds for the controller. When the controller was run on the Embedded Control System only the *input* and *output* signals were inspected. Additional signals can also be monitored if special code blocks are added. Special care is taken so that the monitoring of these signals will not influence the real-time behavior of the ECS. Currently slack-time cannot be measured yet. (Either on the HILS or ECS).

As proof of concept, first a simulation of the controller with the real plant is carried out and as second step the real plant is replaced by a simulation of the Linux setup on the HIL Simulator PC. The controller uses the same hardware and software in both cases. Only the external I/O connection cable will be replaced from the real setup to the simulation PC. The controller software is the same and experiences no differences.

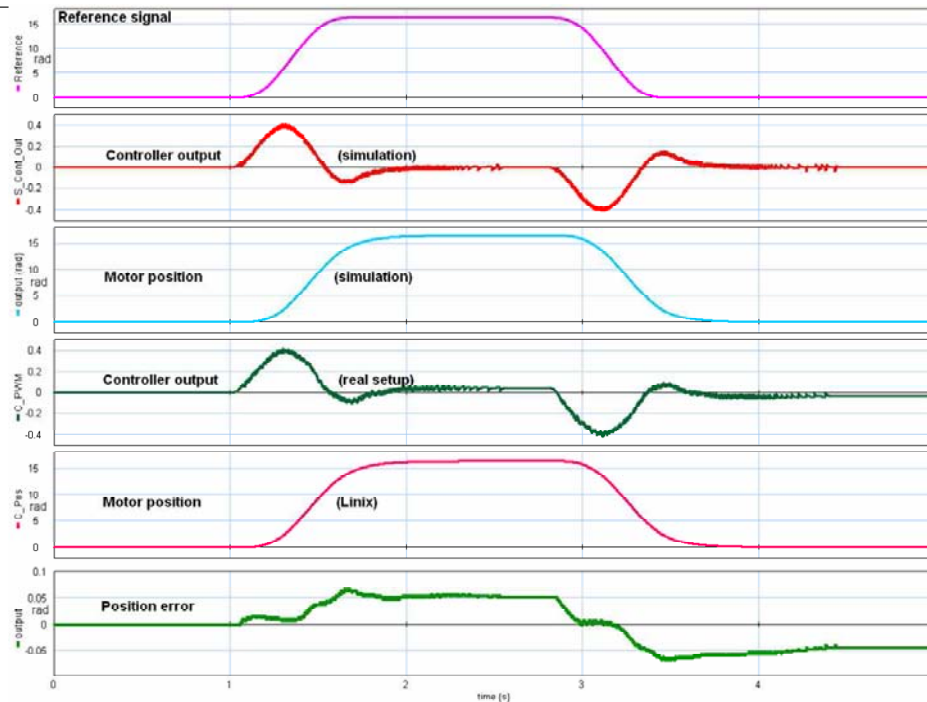


Figure 13: Comparing the simulated Linux with the real Linux

Simulation versus real setup

The first experiment has been performed to compare the 20-SIM simulation results with the results from the real plant using code generation for the controller. Figure 13 shows the course of the PWM and encoder signals when applying a motion profile as reference signal. This test is performed with a sample rate of 1 kHz for the controller. Euler was used as the integration method.

The simulation signals are comparable with the measured signals on the real setup. A small steady state error exists on the position of the real plant. This is due to an unmodeled dead-zone in the PWM steering of the real motor. When the controller output (PWM duty cycle) is smaller than $\pm 0,06$, the Linux motor does not turn.

Simulation versus HIL simulation

The second test has been performed to compare the 20-sim simulation results with the HIL simulation results. Code has been generated for both the controller and the plant and the input and output signals were logged and imported into 20-SIM.

Figure 14 shows a comparison between the 20-SIM simulation results and the ECS-HIL simulator results. The most important difference between simulation and HIL simulation is the physical I/O interconnection. A comparison between the 20-SIM motor position and the HIL simulator motor position (position error line in Figure 14) shows that the PWM I/O and encoder I/O ports are accurate enough for the HIL Simulation. The

error is almost zero and shows only some quantization noise.

The tests have been performed with the controller running at 1 kHz and the HIL Simulator running at 10 kHz. The HIL Simulator runs at a 10 times higher sample frequency for better emulation of the continuous-time system (the real plant has its current state always available, it determines it infinitely fast). The controller and the HIL simulator are not synchronized, i.e. there is no common master clock. The controller computation scheme used here is the measure→steer→calculate method, for both the controller and the HIL simulator to get a precisely periodic steering. This means that at every sample, first a value is measured and the resulting steering value of the previous sample returned before the new value is calculated.

The results from both tests show that the principle of Hardware-In-the-Loop Simulation is working. More simulations and measurements need to be performed, e.g. to check whether there is a possible delay at the HIL simulation outputs compared with the real plant status. One sample delay can introduce a significant error in the position calculation. For example, if the HIL simulator is running at 10 kHz and a the wheel is rotating with an angular velocity of 10 m/s, one sample delay (0,1 ms) will cause an error in the position 1 mm. Experimenting with encoder resolutions, PWM frequencies and controller and HILS sample rates are also desired.

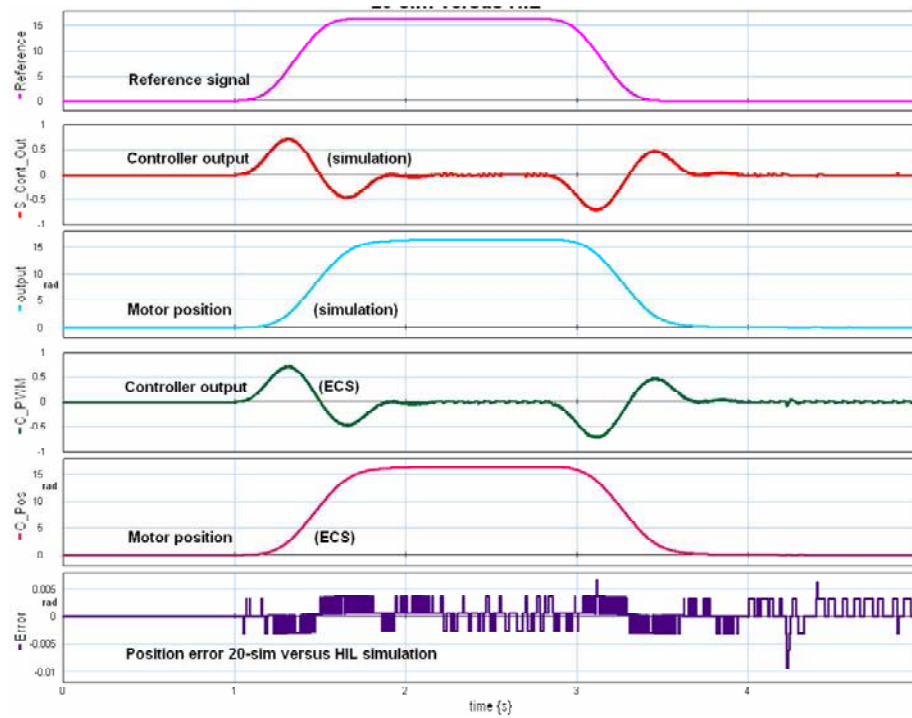


Figure 14: Comparing Simulation and HIL Simulation

VI. CONCLUSIONS AND RECOMMENDATIONS

Hardware-in-the-loop simulation is a valuable aid in design support.

Running the **real** controller on the HILS, before the prototype stage, shows that it functionally works and helps with timing and performance measurements. This is less complex than WCET methods.

Design changes and parameter variation can be made in hours/days instead of weeks/months.

Using FPGA's as configurable versatile I/O devices works and is a cheap means to performing a range of tests.

The HIL simulation approach as in the proposed design trajectory allows for *concurrent engineering* in an early phase.

Recommendations:

- Refining the design trajectory of [1] in order to optimally benefit from HIL Simulation. This requires analyzing a more complex case example.
- More testing of different I/O configurations to check differences in details of behavior and to precisely indicate in which situation a specific HIL Simulation can benefit.
- Add performance measurements means in the automatic code generation trajectory to both HILs and ECS
- Add online visualization and change of parameters.

REFERENCES

- [1] J. F. Broenink and G. H. Hilderink, "A structured approach to embedded control systems implementation", In M. W. Spong, D. Repperger, and J. M. I. Zannatha, Eds., 2001 IEEE International Conference on Control Applications. México City, México, 2001.
- [2] R. Isermann, J. Schaffnit, and S. Sinsel, "Hardware-in-the-loop simulation for the design and testing of engine control systems," Control Engineering Practice, vol. 7, pp. 643-653, 1998.
- [3] B. P. Douglass, Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks, and patterns: Addison Wesley Longman Inc., 1999.
- [4] G. H. Hilderink, A. W. P. Bakkers, and J. F. Broenink, "A Distributed Real-Time Java System Based on CSP", The third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2000. Newport Beach, CA, 2000.
- [5] RTAI, "DIAM RTAI - Real-time Application Interface" <http://www.rtai.org>, 2004.
- [6] E. Andersen, "uClibc website" <http://www.uclibc.org>, 2004.
- [7] B. Orlic and J. F. Broenink, "Real-time and fault tolerance in distributed control software", In J. F. Broenink and G. H. Hilderink, Eds., Communicating Process Architectures 2003. Enschede, Netherlands, 2003.
- [8] H. Kopetz, Real-Time Systems, Design principles for Distributed Embedded Applications. Boston: Kluwer Academic Publishers, 1997.
- [9] H. Kopetz and R. Obermaier, "Temporal composability," IEE Computing & Control Engineering Journal, vol. 13, pp. 156-162, 2002.
- [10] Mesa Electronics, "Mesa Electronics" <http://www.mesanel.com>, 2004.
- [11] CLP, "20-SIM" <http://www.20sim.com>: Controllab Products, 2002.