Abstract – The modeling and analysis of Cyber-Physical Systems (CPS) is one of the key challenges in complex system design as heterogeneous components are combined and their close interaction with the physical environment has to be considered. This article presents a methodology and an open toolset for the virtual prototyping of CPS. The focus of the methodology is the virtual prototyping of the embedded software combined with the prototyping of the physical environment in order to capture the complete closed control loop of the software over the hardware via sensors/actors with the physical objects. The methodology is based on the application of integrated open source tools and standard languages, i.e., C/C++, SystemC, and the Open Dynamics Engine, which are combined to a powerful simulation framework. Key activities of the methodology are outlined by the example of an electric two-wheel vehicle.

I. Introduction

Cyber-Physical Systems (CPS) are complex heterogeneous systems with a tight combination and interaction between the computer-based system and physical elements where physical characteristics effect computation and vice versa [16][17][34]. A CPS is typically composed of different types of components as given in Fig.1:

- sensors and actors interacting with a physical environment in a closed control loop, e.g., acceleration, gyroscope sensors and electrical motors;
- power supply, like a battery in the case of a mobile system;
- analog hardware components, like power electronics;
- digital hardware components, like digital converters;
- different types network for the communication of the different components, like CAN, MOST, or FlexRay;
- homogeneous and heterogeneous software microprocessors/controllers or cores for the execution of software (application software, real-time operating systems, and hardware-dependent software).

Due to this classification, CPS can be recognized in our daily life in various forms with different complexities, like smart phones, automotives, robots, and assembly lines. As such, the mixed HW/SW components of a CPS can be distributed over several bodies, boards, or just resides on a single chip.

This indicates the inherent complexity when we have to cope with the design of a CPS as types and heterogeneity of the components can vary between different domains and applications.

There are currently very few approaches, which address the design of a CPS in its entirety. Jensen et al. [15], for instance, presents a model-based design methodology with the focus on physical properties by the case study of a tunneling ball device. However, several design flows, are still based - with some extend - on the classical principles of HW/SW codesign [29]. In dedicated domains, like automotive system development, the design of a Cyber-Physical System follows the general principles of the classical V-model [28], i.e., after a specification, the system is partitioned into the design of the different subdomain models, which are latter incrementally integrated and tested. Therefore, partitioning and integration with respect to different languages and tools from multiple domains is essential in such a design process. In this area, we can find approaches and toolsets, which are based on Matlab/Simulink design flows [33][8]. In this context, it was recognized that the early availability of multi-domain/multi-language simulation/emulation frameworks is mandatory. Thus, meanwhile, several EDA tool vendors support virtual prototyping for embedded software development in multi-language simulation environments [6][18][31]. However, the close interaction with virtual physical environments for a holistic design view is still not really captured.

In this paper, we present a methodology and open toolset for the virtual prototyping of CPS. We focus on two different
virtual prototyping aspects. These are the virtual prototyping of the physical environment (rigid bodies and kinematics) and its integration with a mixed HW/SW simulation/emulation environment for the virtual prototyping of the embedded software in network based applications. The toolset is solely based on the integration of open tools and libraries: Open Dynamics Engine (ODE) [21], SystemC/SystemC-AMS [12][22], and QEMU [25]. We selected those tools as they are available as open source and cover a wide spectrum of subdomains for CPS. As they are all based on C/C++, they also provide an open interface to complementary design flows with C code generation, like Matlab/Simulink [17]. In that context, we discuss the virtual prototyping of embedded software with focus on the refinements of hardware-dependent software and introduce a refinement process with fast simulation/emulation support and its integration with a virtual physical environment.

The remainder of this article is structured as follows. The next section discusses related work in the area of mixed simulation and virtual prototypes. Thereafter, we present our refinement process and the integration with the virtual physical environment. Section IV, gives an overview of our experimental results before the conclusion closes the article.

II. Related Work

One of the first works in the area of mixed system simulation was provided by Ptolemy [5] in 1992. Later, Ptolemy II [9] generalized those concepts and extended the set of models of computation, like the continuous time domain for mixed signal simulation. At that time, most of the HW/SW co-design approaches [29] considered software mainly in combination with digital electronic systems partly combined with FPGA emulation.

However, the low execution speed often limited the complexity of the analyzed system in the early years. With the outcome of C based system description languages like SystemC [25], mixed HW/SW simulation became significantly faster, so that more complex systems could be addressed. With the introduction of SystemC-AMS (Analog and Mixed Signal) [12], it also becomes applicable for mixed digital/analog simulation.

At that time, also higher abstraction levels were introduced, like the SystemC based TLM standard [23] and refinements of system level abstraction by application level, task level, firmware level etc. [26]. As such, considerable results could be achieved to increase the simulation speed, which thus also increasing the complexity of the considered system [27][36]. In turn, abstraction in combination with faster simulation hosts also allowed an increasing level of detail for embedded software so that complete software systems (application software, middleware, and hardware-dependent software including the operating system) could be cosimulated with a software emulation. Such virtual prototyping platforms for embedded software support multiple instruction sets and are typically based on emulation acceleration by binary translation like by QEMU [25], Simics[35], and OVP [14].

Complementary to the virtual prototyping of embedded software, we can find several approaches for combining simulation with Virtual/Augmented Reality environments in the area of mechanical engineering to evaluate early virtual mock-ups, which can be modeled in sufficient details by means of 3D CAD tools, like AutoCAD. A comprehensive overview of systems, which combine Virtual/Augmented Reality with simulation can be found in [10]. There exist also some dedicated solutions in combination with Matlab/Simulink [11]. In the context of Matlab/Simulink, some commercial tools are available, which provide interactive 3D environments for simulation, like dSPACE MotionDesk [8] for automotive software testing. Very few approaches already report multi-domain cosimulations like [1], which introduces a cosimulation framework with Matlab/Simulink, FPGAs, and SystemC in combination with an OpenGL-based 3D user interaction. However, that work mainly investigates the principles of a multi-domain execution platform based on the application specific integration dedicated to the RailCap shuttle system and is not applicable as a general CPS development platform.

III. Virtual Prototyping of Cyber-Physical Systems

Our virtual prototyping methodology is based on a set of open source tools and libraries. The next paragraph introduces the applied tools after which the methodology is outlined and details on the virtual prototyping are given.

A. Tools

To achieve a tight integration, our prototyping environment is solely based on open source tools and libraries: SystemC(-AMS) [22] for mixed digital/analog systems and as a simulation integration framework, QEMU [25] for fast software emulation, and the Open Dynamics Engine (ODE) [21] for the simulation of the physical environment.

SystemC. IEEE 1666-2005 SystemC is a C++ based system level description and simulation language. SystemC basically extends C++ by structural entities, namely the SC_MODULEs with embedded processes that are executed by a pseudo-parallel simulation. The reference implementation of the SystemC simulator comes as a freely available C++ library with header files describing the classes and a link library that
contains the simulation kernel. A SystemC description can be compiled by any ANSI-compliant C++ compiler. The resulting executable specification realizes an event based simulator that allows high speed simulation with integrated simulation control facilities. For a more detailed introduction to SystemC, the reader is referred to standard literature given at [22].

**QEMU.** QEMU is a software emulator, which provides an instruction and register accurate CPU abstraction with the support of multiple instruction set platform, e.g., x86, ARM and PowerPC. QEMU comes with an efficient dynamic binary translation based on a fetch/decode/execute cycle. The cycle applies an online compilation of instructions into the Instruction Set Architecture (ISA) of the emulation host. QEMU can operate in two modes: (i) user mode and (ii) full system mode. The user mode supports an efficient user space simulation of a single process on top of one process on the simulation host. In user mode, a system call raises an exception, which in turn invokes a corresponding host OS system call. The user mode maps the target memory directly to the host memory, so that it abstracts from any specific memory model. The QEMU full system mode includes an entire target platform with full I/O and kernel space access for operating system and driver simulation so that system calls can be directly executed. Even though not initially planned for a usage in virtual prototyping, the dynamic binary translation engine of QEMU is very efficient, and several approaches have been proposed for integrating it with SystemC [2][19], as it gives the high level of modularity the electronic system level designs require.

**ODE.** The Open Dynamics Engine (ODE) is an open source library for simulating physical rigid body dynamics with arbitrary mass distribution for use in interactive real-time simulation. ODE is platform independent with an easy to use C/C++ API. The equations of motion are derived from a Lagrange multiplier velocity based model using a first order, highly stable integrator. In ODE, bodies of various shapes can be easily defined and connected with joints of various kinds. ODE emphasizes speed and stability over physical accuracy unless the step size is small. For increased accuracy, a 64 floating-point mode is available, which we applied for our studies. Additional libraries, like the Inventor Physical Simulation API (IPSA), provide a 3D representation and interaction of bodies at the user interface. As such, interaction between bodies, like collision detection, can be easily captured, controlled, and monitored.

**B. Multidomain Design**

The entire design process of complex CPS has to consider a holistic view over all domains to cover all effects in the context of the physical environment. As such, we can roughly divide the components into software, electronic hardware (analog and digital), sensors, and actors (cf. Fig.2). In this classification, though they typically come as electric and electronic hardware, we consider sensors and actors separately as they are the main interface for interaction with the physical environment and thus require special attention in the design process.

![Diagram](https://via.placeholder.com/500)

**Physical Environment.** The virtual physical environment can be refined from the physical object and their properties (like size, weight) and the corresponding physical laws (like gravity). Due to the nature of our applications, we are mainly interested in kinematics of rigid bodies in motion. However, similar design principles can be applied when an alternative engine for the adequate virtual representation of other matters with different behavior, like fluids, becomes available. The identification of all relevant physical objects and values is mandatory as the adequate representation of physical objects and their interaction with the mixed HW/SW system via sensors and actors is of utmost importance for the correct implementation of closed-loop functions. Even slight inaccuracies in the closed-loop may results in major malfunctions. At this stage, different alternatives in weights and shapes and their impact on the selection of sensor and actors can be easily captured and evaluated.

**Sensors & Actors.** Selection and dimensioning of sensors and actors has to be determined in combination with the physical properties of the physical bodies, which have to be monitored, controlled, and moved. At this stage, different selection of sensors and actors can be evaluated for different alternatives of physical bodies. The size and the shape of a body, for instance, have an impact on the number of sensors and their placement. An example is the evaluation of required number and types of distance sensors, which can be evaluated at different heights and locations of a vehicle where distance sensors vary in directivities, frequencies, response times, power consumptions, weight, dimensions, and price. The weight of a body, for instance, additionally determines the required size of a motor. When not already available, abstract models of the sensors and actors have to be implemented. As
they are typically reused, the implementation can be derived from existing data sheets.

**Electronic Hardware & Software.** Starting from a functional description, the system is partitioned into its hardware and software components, which are later modeled, analyzed, parameterized, implemented, and tested separately. However, even for medium sized system the codevelopment of the subcomponents is crucial as CPS typically come as distributed systems with distributed software running on different processors connected by a network. In this context, virtual prototyping of software requires the early execution of the software binary code on the target processor (i) to start software development (application software and hardware-dependent software) as soon as possible and (ii) to identify compiler-specific effects on the target platform and their impact on the behavior of the complete system. Though virtual prototyping environments can already run the target software binaries at a considerable speed, the virtual prototype execution of a complete system is still often too slow so that abstractions have to be applied like for bus communication and operating systems, which can give simulation accelerations by factors up to 40,000x [36] as we will see in more details in the next chapter.

As we are mainly interested in refinement of the electronic parts and the software rather than on the design of mechanical and electric parts, we focus on the refinement of the software in combination with the hardware in the remainder of this chapter. As its integration with the physical environment is always application specific, we outline the integration by an example in the section thereafter.

**C. Virtual Prototyping of Embedded Software**

For our refinement in the context of a SystemC based simulation environment, we presume a functional model, which divides the system into communicating hardware and software components. We further presume the availability of the different components as C/C++ based models or source code, which can be assigned to parallel executing simulation units, i.e., SystemC modules. Their communication structure can be roughly determined as an abstraction of the interconnection network, which is subject for further refinement. Software components have to available as C source code or as object code, which can be embedded into or linked with SystemC processes. The source code may be of arbitrary complexity, e.g., it can be generated from other toolsets, like by Targetlink [8] from Matlab/Simulink.

In this scenario, time critical requirements, like sampling rate or latency, or resource constraints, like limited numbers of microcontrollers, may require the application of a (real-time) operating system, which provides task scheduling and protocols for shared resource access. Additionally, low level software is required like communication stacks for network communication and specific drivers, like I/O to sensors and actors. For simplicity, we denote low level software in combination with operating system as hardware-dependent software (hds) in the remainder of this article.

For virtual prototyping with fast simulation support, we introduce three different refinement levels: native software, binary software, and microprocessor level (cf. Fig.3), which are introduced in the following paragraphs.

![Fig.3. Refinement of Embedded Software.](image)

**Native SW Level.** In a first step, software is considered at source code level where functions of the application software are assigned to tasks, which are subject of parallel execution and scheduling [3]. For an efficient analysis, the software is natively compiled for direct execution on the simulation host and linked to the simulation environment, which is determined by a SystemC simulation environment. In order to achieve fast simulations, abstractions of the operating systems and the hds are applied like as they are implemented by our aRTOS SystemC library [36]. For this, we assign software task to an aRTOS operating system (OS) context, which compare to abstract processors with an individual task and interrupt handler. For task execution, a canonical interface for the emulation of system calls supports basic OS functions, like activating and deactivating tasks.

For execution time simulation, the software source code of each task is divided into segments, which are annotated with execution times. The execution times are determined by either a static code analysis or by runtime measures on the final target processor. During simulation, the execution times of segments are dynamically accumulated and communicated to the corresponding OS context at the end of each segment. Based on this information, the task scheduler of the OS context determines the next task for execution. The interrupt scheduler of an OS context manages interrupts and executes the corresponding Interrupt Service Routines (ISR). I/O communication at this level, like to sensors and actors, is abstracted by basic communication channels like FIFOs.

Through those abstractions, we can reach considerable simulation speeds for early functional, timing, and power analysis and for design space exploration. Experiments have
shown that the simulation at native software level with (RTOS) abstraction is 2,000x-40,000x faster compared to classical instruction set simulation with a loss in accuracy of 1%-10% compared to a logic analyzer [35].

**Binary Software Level.** At this level [2], the software is compiled with the compiler of the target platform. This supports already a most realistic analysis as all compiler features and anomalies can be considered like code optimization and compiler bugs. As the target platform is typically different from the simulation platform, we apply abstraction techniques in combination with efficient software emulation. In our virtual prototyping environment, we have combined advantages of user mode QEMU with our aRTOS SystemC library by integrating QEMU user mode emulators into an aRTOS OS context, which requires a few modifications in the QEMU execution cycle. For fast simulation, system calls of the application software, which runs in user mode QEMU, are redirected to aRTOS calls without modifying the final binary code. For efficient binary task execution, QEMU applies dynamic binary translation. For this, the binary code is divided into Basic Blocks (BB), i.e., linear code segments with a final branch instruction at the end. QEMU uses a dynamic code generator to translate BBs at run-time by concatenating precompiled simulation host code segments, i.e., Translated Basic Blocks (TB). For faster execution, TBs are stored in a TB cache. Then, the major emulation effort is just looking up and executing TBs from the cache. For timing and power analysis, TBs have to be annotated by time and power information, which are elaborated by a dynamic code analysis at compile time.

As such, considerable execution speeds can be reached. Our experiments demonstrated that the execution speed becomes just 1.5x-10x slower compared to the native software level although the target binary is already executed. As the analysis is performed on the target instruction set on the virtual CPU platform, in contrast to the native software level, compiler optimization and dynamic CPU effects like delays due to data dependencies can be already investigated.

**uProcessor Level.** At this level, hds abstractions are replaced by final target binaries. This implies the installation of the final operating system including communication stacks and drivers. For virtual prototyping at that level, we apply full system mode QEMU. In the SystemC model, this requires the replacement of the aRTOS SystemC library calls by TLM transactors for time- or cycle-approximate simulation integration of a virtual QEMU microprocessor.

**D. Integration with the Virtual Physical Environment**

As the modeling of the virtual physical environment and implementation of drivers for sensors and actors is most application specific, we demonstrate this by a use case, the re-engineering of an electrical two-wheel vehicle (Elektor Wheelie), which compares to the well-known Segway Personal Transporter. The vehicle is operated by a driver through weight balancing moving the vehicle forward and backward. Left and right directions are controlled by the steering bar in the middle. As the vehicle is originally equipped with a microcontroller, open source software, several sensors, and electric motors and tightly interacts with the driver in real-time, we can consider it as a true Cyber-Physical System.

The two wheels of the vehicle are driven by two 500W DC electro motors where power is supplied by two 12V lead AGM batteries. The tilt movement of the driver for speed control is taken by a gyro sensor (Gyro IDG300) and an acceleration sensor (ADXL320). The steering bar is controlled by a simple rotary potentiometer. The original vehicle board is equipped with an Atmel ATmega32 microcontroller, which runs with a sampling rate of 10ms and controls the motors via an H-bridge, which is connected by PWM to the processor. The original software is open source and implemented in Basic (BASCOM) and comes without any operating system. Software device drivers are provided by the BASCOM environment.

For an extension by additional distance sensors, we completely exchanged electronic hardware and software. For this, we replaced the ATmega32 processor by an AT90CAN128 and connected it to two additional microcontrollers by a CAN network (1xAT90CAN128, 1xARM9) for the integration of the additional sensors and servo and the separation of the main control task (cf. Fig. 5).

![Fig. 5. Extended Hardware Architecture.](image)

As the modeling of the virtual physical environment and implementation of drivers for sensors and actors is most application specific, we demonstrate this by a use case, the re-engineering of an electrical two-wheel vehicle (Elektor Wheelie), which compares to the well-known Segway Personal Transporter. The vehicle is operated by a driver through weight balancing moving the vehicle forward and backward. Left and right directions are controlled by the steering bar in the middle. As the vehicle is originally equipped with a microcontroller, open source software, several sensors, and electric motors and tightly interacts with the driver in real-time, we can consider it as a true Cyber-Physical System.

The two wheels of the vehicle are driven by two 500W DC electro motors where power is supplied by two 12V lead AGM batteries. The tilt movement of the driver for speed control is taken by a gyro sensor (Gyro IDG300) and an acceleration sensor (ADXL320). The steering bar is controlled by a simple rotary potentiometer. The original vehicle board is equipped with an Atmel ATmega32 microcontroller, which runs with a sampling rate of 10ms and controls the motors via an H-bridge, which is connected by PWM to the processor. The original software is open source and implemented in Basic (BASCOM) and comes without any operating system. Software device drivers are provided by the BASCOM environment.

For an extension by additional distance sensors, we completely exchanged electronic hardware and software. For this, we replaced the ATmega32 processor by an AT90CAN128 and connected it to two additional microcontrollers by a CAN network (1xAT90CAN128, 1xARM9) for the integration of the additional sensors and servo and the separation of the main control task (cf. Fig. 5).
the front of the vehicle. The bar is kept in its horizontal position by a servo motor for correct horizontal alignments of the sensing direction.

Finally, we re-implemented the software in C and separated different functions into software tasks, which are scheduled by a basic real-time operating system. For the Atmel microcontrollers, we implemented a sensor, an actor, and a communication task. The central ARM processor executes the main vehicle control task and a communication task for interacting with the CAN bus.

For the virtual physical environment, we have (i) to model the virtual bodies and (ii) to implement software drivers for sensor and actor interaction. For the correct interaction, we also have to consider side effects of the power supply, which are outlined in the following paragraphs.

**Virtual Bodies.** We first abstracted the physical environment by six bodies plus test surface objects:

1. vehicle chassis (covering motors and batteries)
2. sensor bar
3. two wheels
4. steering bar (composed of two fixed bars)
5. driver

The bodies were pair-wise connected by rotary axes. However, it soon turned out that the abstraction of the driver’s body by a cube is not sufficient for operating the vehicle. Therefore, we had to divide it into head, torso, arms, and legs to correctly model the tilting movement, which can be controlled by an external joystick. We defined the individual bodies and masses comparing to an average adult of 175cm height with 80kg. However, those properties can be easily exchanged and assigned to other requirements. Fig. 5 shows a screenshot of the OpenGL based graphical 3D user interface with driver, vehicle, and test environment. At the front of the vehicle, we can see an additional visualization of the range of the distance sensors.

**Sensors.** After having attached actors and sensors to a position on the virtual bodies, software drivers have to be implemented for their interaction with the simulation environment. Those drivers have to convert physical values from the virtual environment to analog or digital (voltage) signals, which can be processed by simulation. After the conversion they have to be compliant with the digital values format provided by the A/D converters. The gyro sensor IDG300, for instance, delivers 2mV/°s offset voltage to 1.5V dependent on the angular velocity, which has to be further converted into digital 10bit values. Hereby, also the 100us latency of the A/D converter has to be considered for a time accurate simulation. Due to the individual functions, different abstractions and approximations can be applied. It is basically also due to the individual application to model additional effects, like sensor noise.

**Actors.** Our experiments have shown that the implementation of software drivers for specific actors like motors require special attention when power consumption is covered in the model. In our case, the motor model is controlled by a 64kHz PWM signal and requires the torque on the axes as an input at each simulation step. In this context, for a realistic simulation, side effects like internal resistance, inductivity, and the rotation speed of the motor have to be considered. An important factor is also the voltage induced by the rotation of the motor, which has also an impact on the torque.

**Power Supply.** Our virtual physical model at the introduced level of detail already supports power consumption analysis. Based on several measurements with different loads for voltage drops of the batteries, we derived an abstract battery model, which turned out to be sufficiently precise to make power consumption estimations. In such models, consumption details of the motor control are considered like the alternating switching of the motors, which reduces the voltage drop at the battery. Additionally, this is also increases the torque, which may result in an unstable balance control when this effect is not considered, which in turn indicates again that a holistic view is mandatory for CPS closed-loop controller design.

For monitoring various data at run time, we also have implemented an additional LabVIEW panel, which is shown in Fig. 7. The panel provides different instruments for battery control (upper right), motor/power control (upper middle), and sensor data (upper left), as well as two voltage graphs for the sensors and the battery (lower left).
IV. Experimental Results

We performed several experiments through the last years on multiple target platforms like PowerPC, Atmel, and ARM for simulation acceleration.

Schirner et al. [26] have published experimental results for SpecC simulation speed on different abstraction levels. They report a simulation speed-up by 1,000x-10,000x from native software level to instruction set simulation with a simulation error of ca. 2%. Our aRTOS library in [35] basically compares to their SpecC canonical operating system abstraction in [26]. In our experiments, we found simulation speed-ups of 2,000x-40,000x with an accuracy of 1%-8% compared to a logic analyzer (cf. Fig. 8). These numbers also meet our daily experience with the application of aRTOS in the area of scheduling and interrupt analysis.

V. Summary and Conclusions

In this article, we introduced a methodology and SystemC-based open toolset for the virtual prototyping of Cyber-Physical Systems. The toolset is solely based on the integration of open C/C++ based tools and libraries: SystemC/AMS, Open Dynamics Engine (ODE), and QEMU. The tools cover a wide range of domains so that we can support a holistic view required for Cyber-Physical Systems design. We showed that both, the virtual prototyping of the embedded software and the virtual prototyping of the physical environment are important for the development of closed-loop control application. In this context, we recognized that the adequate implementation of software drivers for virtual sensors and actors is of utmost importance as minor inaccuracies and incompleteness in the model may have a major impact to the control loop.

Current experiments gave promising results with respect to fast simulation speeds. The application of the toolset in the context of our case study demonstrated that native SW level and binary SW level are both adequate for real-time interaction with the virtual physical environment. However, as we are finally interested in a full system mode simulation in combination with the physical prototype, we still need to improve the performance of the SystemC/QEMU integration at that level. Investigation in this direction also consider TLM 2.0 based strategies, i.e., loosely-timed, approximately-timed, for a more efficient QEMU integration.

Acknowledgments

This work was partly funded by the DFG Collaborative Research Centre 614 and by the German Ministry of Education and Research (BMBF) through the BMBF project SANITAS (01M3088) and the ITEA2 projects VERDE (01IS09012), and TIMMO-2-USE (01IS10034A). We also appreciate the fruitful discussions and the support of Henning Zabel who is now with Beckoff Automation GmbH.
References


