Simulation of Model Execution for Embedded Systems

Jörg Christian Kirchhof
Software Engineering
RWTH Aachen University
Aachen, Germany
kirchhof@se-rwth.de

Evgeny Kusmenko
Software Engineering
RWTH Aachen University
Aachen, Germany
kusmenko@se-rwth.de

Jean Meurice
Software Engineering
RWTH Aachen University
Aachen, Germany
jean.meurice@rwth-aachen.de

Bernhard Rumpe
Software Engineering
RWTH Aachen University
Aachen, Germany
rumpe@se-rwth.de

***Abstract*—In automotive and robotics, simulation is an indispensable tool for testing and validation. A simulator is able to test a system under varying conditions at low cost in a non-safety-critical environment. Furthermore, in the development process of a new vehicle, the first prototype is mostly produced after a long design phase, which can take up to several years. Before the prototype is available, test can only be performed in a simulation. To make the simulation results reliable, both the system and its environment need to be modeled as realistic as possible. As modern vehicles include a large amount of software, the execution of vehicle software needs to be simulated with respect to the underlying E/E infrastructure. In this paper, we present a simulation framework for the execution of vehicle control models deployed in a vehicle network. Furthermore, the execution simulator is embedded into a vehicle simulator, making it possible to validate the vehicle software functionality under the given hardware conditions.**

***Index Terms*—simulation, automotive, E/E infrastructure, model execution,**

I. Introduction

Software for embedded systems, especially for automotive applications and Cyber-Physical System (CPS), is getting increasingly more complex. This is, partly, due to the increasingly challenging tasks these systems have to solve. Future cars are expected to include complex functionalities such as autonomous and cooperative driving. As software gets more complex, it also gets harder to develop and test. Models can be used to abstract from this complexity and, thus, make the development process both simpler and more reliable. However, the changing environmental influences still require the software to be tested in a large number of different scenarios to ensure its reliable execution. Due to time and resource constraints, the software cannot be tested solely on prototypical hardware. Moreover, the hardware may not even be developed before the software.

Simulators allow to effortlessly test software in a large number of scenarios and under a multitude of environmental influences. As simulators only *simulate* the real world, they make abstractions from the real world. As a result, the results calculated by simulators are expected to deviate to a certain extent from the results that would be observed on a similar setup in the real world. Small deviations are expected and can be tolerated. However, strong abstractions from the real world can lead to imprecise results [1]. This is especially important in autonomous and cooperative driving applications as an error of the system might not only cause large monetary damages but also threaten human lives. Thus, an accurate prediction of the system's behavior is crucial for a simulation.

An important aspect of accurately simulating the behavior of a system is the simulation of its time behavior. Embedded systems, such as Electronic Control Units (ECUs), are often resource-limited, especially compared to the computers used to develop the software they execute. If the system is not capable of executing the desired functionality within certain time limits, results may often be worthless. For example, if a system used to trigger the brakes in an autonomous car is too slow, the car might crash.

Existing solutions either offer a precise prediction of the embedded system's time behavior at the cost of a time-consuming simulation or require developers to model the system-under-test within the simulation environment, hence inducing a source for hard to find errors. In this paper, we present a method for integrating a hardware emulator of the target platform into a general-purpose driving simulator responsible for simulating the environmental influences of and interactions between vehicles. This allows us to combine the efficient simulation of the MontiSim simulator with the accurately predicted time-behavior of a hardware emulator. The control units within the MontiSim simulator are defined using the EmbeddedMontiArc (EMA) language. Thereby, we leverage the benefits of a model-driven development process. At the same time, using the binaries compiled from the source code generated from the EMA models as input for the emulation allows us to both circumvent the time-consuming creation of a second model solely meant for simulation and eliminate the source of potential errors induces by this task.

The remainder of this paper is structured as follows. Sec. II introduces EMA, MontiSim, and introduces terminology. Sec. III lists the requirements for our simulator. Following that, Sec. IV describes our hardware emulator and its integration in MontiSim in detail. Our solution is then evaluated in Sec. V. Sec. VI presents related work and Sec. VII concludes the paper.

II. Background

The models for which the proposed solution provides a simulation environment are created using the EmbeddedMontiArc (EMA) language [2], [3]. This is a modelling language used for describing the architecture of components and their communication. A component has a set of input and output ports (typed variables) and can have a set of sub-components. Components use these ports to exchange values with other components. Therefore, ports can be connected to other ports using connectors. A component "cycle" (an update) uses the current state of the input ports to update the output ports. This happens through computations or the use of its sub-components. The port system supports generic, array and vector types, all of which are strongly typed. EMA models are used to generate C++ code, which is then compiled using standard compilers. Furthermore, the EMA language family contains extensions allowing the integration of, e.g., equation solvers or neural networks with their specific language extensions.

The simulation environment MontiSim is an autonomous vehicle simulator [4], [5]. Its goal is testing autopilots and other vehicle software developed using the EMA language and its extensions. It is capable of loading a simulation described using a *simulation language*. This language describes the environment of the simulation, i.e., the map, the weather conditions, time of day, etc., as well as the set of vehicles that are to be simulated. The map on which the vehicles drive is imported from OpenStreetMap [6] data.

The vehicles are configured with start and destination coordinates, a physical vehicle model and an E/E setup. This E/E setup contains the different ECUs and their software alongside sensors, actuators, and communication buses.

Prior to the presented simulation integration, the MontiSim simulator had no means of representing the execution time of vehicle software within the simulation. This is problematic as soon as the execution time of the software surpasses the tick duration of the simulator, in which case the software gains an unrealistic computation power in regards to the simulated time. This is a critical missing aspect of the software simulation, especially for automotive applications. Therefore, the proposed simulator fills this gap by representing the computation time of software inside the MontiSim simulator.

The proposed simulator is mainly used as sub-simulator inside the MontiSim simulator. To differentiate between the simulation levels, the following terminology will be used:

Emulator The proposed simulator that emulates the behavior of computer hardware.

Emulation The virtual environment in which the emulated software lives.

Simulator The overarching simulator using the computer hardware simulator inside its simulation.

The difference between *emulation* and *simulation* is that a emulation computes the underlying behavior of a system while the simulation only provides the visible effects of such a system. The proposed sub-simulator is nearer to the emulator concept than the MontiSim simulator, which motivates this denomination.

III. Requirements

Following the goals of the MontiSim simulator as well as those of EMA models, the emulator aims at fulfilling the following requirements:

R1 Correct software behavior: the emulator must reproduce the real logic behavior of the emulated software. The code emulation must yield the same outputs as when executed on real hardware.

R2 Time evaluation: the emulator must evaluate the execution time of the emulated software so that this critical aspect can be represented in the overarching simulation.

R3 Reproducibility of the simulations: given a simulation description and EMA models (the emulated software), the simulation must yield the same output independently of the platform and hardware used.

R4 Platform-independence: it must be multi-platform in the first place and allow the emulation of software compiled for other platforms.

R5 Variability of the hardware models: the properties of the emulated hardware running the EMA models must be configurable. This includes, e.g., the CPU model, its frequency, the memory properties, etc.

R6 Generic emulator: the hardware emulator can be used to emulate any program and evaluate their execution time.

IV. Hardware Emulator

The proposed simulator will be referred to as the *Hardware Emulator*. This emulator contains a program emulating component as well as models for the evaluation of the execution time of the program. The Hardware Emulator does not run an operating system in the virtual computer but rather emulates its functionalities from outside the emulation.

The following sections will describe the components of the Hardware Emulator and their functionalities.

A. Unicorn Emulator

The program emulation capabilities of the Hardware Emulator come from the existing *Unicorn* emulator [7]. This emulator manages registers and virtual memory around a *QEMU* core [8]. The engine is capable of emulating the behavior of programs under multiple architectures and in different modes. In this case, the 64-bit mode of the x86 architecture is used. The Unicorn emulator is open-source and multi-platform. Since the emulated architecture is independent of the system running the emulator, it is also cross-platform (R4).

The interface of the Unicorn engine allows allocating and managing virtual memory sections. The interface also allows reading and changing the registers related to the used architecture. Finally, it allows starting a single-threaded program execution at an arbitrary memory address. By taking advantage of this fact and by knowing the address of a specific function,

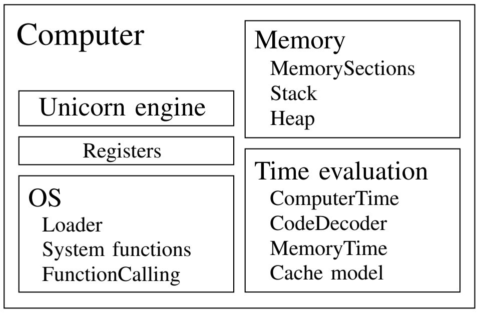

Fig. 1. High-level overview of the main component of the Hardware Emulator.

functions of a program can be executed from outside the emulator. The emulator then reproduces the behavior of program instructions depending on the contents of the registers and the memory (R1).

Finally, the Unicorn engine allows monitoring every instruction execution and memory access. This enables the evaluation of the different execution time models of the Hardware Emulator (R2).

B. Computer Model

The Hardware Emulator is built in a component and sub-component style for logical separation of the different virtual computer parts. The main component is a *Computer* component. Its final goal is to be able to set up a software execution environment, load a program and present an interface to call the functions of this program.

Fig. 1 shows the *Computer* component and the types of sub-components it contains. The memory abstractions include a simplified interface to the registers, an object-oriented representation of the virtual-memory sections, and components representing the program stack and dynamic heap. The OS emulation component is an instance of a program loader, a set of emulated operating system functions and a specification of the function calling standard used under a specific operating system. There are currently implementations of the OS emulation for Linux and Windows.

The emulator is currently built to load only one program. Therefore, the executed programs must be compiled with static libraries (or archives under Linux). Furthermore, the emulator does not implement program unloading. When the lifetime of a specific program ends, the entire *Computer* component is destroyed, including its Unicorn engine instance. Once the program is loaded, there can be an infinite number of calls to its functions and the state will be saved.

However, only the encountered operating system functions, operating system objects, and function argument types are implemented. Thus, new models might require some updates of the emulator.

C. Memory and Register Abstractions

To ease the interaction with the *Unicorn* engine and memory management, a set of components present an object-oriented interface for the original *C* functions of the Unicorn interface.

1) Registers: The *Registers* component allows direct access and type casting to the registers. It also prevents a bug when reading the registers inside any of the Unicorn *monitoring callback*. When trying to read the contents of a register into a buffer that lived on the stack *inside* a Unicorn callback, the entire program would crash. By using a buffer allocated before running the engine, it allows to access the registers inside callback which is critical to emulating the operating system functionalities.

2) Memory and memory sections: The *Memory* component encapsulates Unicorn engine functionalities related to its internal memory system. It is responsible for allocating and managing *MemorySection* objects.

These contain meta-information about the allocated virtual memory sections of the Unicorn engine. This includes the start address and size of sections as well as their reading, writing, and execution permissions. A *MemorySection* also holds debug information such as a section and module name, an optional mapping to the program file if the content of the section originates from the program file, and an annotation system. This annotation system allows adding "Notes" on any memory range. This is used to mark and name symbols, operating system structures, and system handles. The *Memory* component also contains a lookup map to efficiently access the *MemorySection* objects corresponding to virtual addresses.

The start addresses and sizes of the virtual memory sections of the Unicorn engine must be multiples of an internal *page size*. Using the page size, the desired addresses and sizes of memory sections can be padded to multiples of the page size using the following **integer arithmetic**: $start\_padded = (start\_address/page\_size) * page\_size$, $size = end\_address - start\_padded$ and $size\_padded = (((size-1)/page\_size)+1)*page\_size$. For a desired memory range with start address $start\_address$ (inclusive), end address $end\_address$ (exclusive) and $page\_size$ the internal page size of the engine, this will give a padded memory section with start address $start\_padded$ and size $size\_padded$.

To simplify the memory layout of the computer, the different memory sections have predefined ranges in the virtual memory space that are defined inside a namespace for easy arrangement.

To help interact with the emulation memory from the outside, a *SectionStack* component can be attached to any *MemorySection*. This is a minimalistic allocator that works in a stack fashion on the section and does not implement deallocation. This allows for fast laying out of objects in the computer memory. This is used by various other components of the *Computer*.

The last elements of the *Memory* component are helper functions to read certain data types from the engine's memory. Currently implemented are functions to read and write strings of chars and wide chars as well as functions to read multi-byte numbers. The string-reading functions read the memory one by one from a starting address into a buffer until encountering a null character or reading invalid memory. In the latter case, it returns an empty string (where the first character is the null character). If successful, it returns a char pointer to the buffer it used to save the string. The number-reading functions read the number of bytes making up the number from the emulator and cast the result to the correct type.

3) Virtual stack and heap: Virtual *stack* and *heap* components use the memory system to manage memory sections used as stack and heap for the emulated program. The heap component has a memory allocation emulation that can be used by the operating system emulation to dynamically allocate and free memory chunks from the heap. The stack component also presents push and pop functions to interact with the virtual stack from outside the emulation.

D. Operating System Emulation

The emulator does not load a complete operating system image into the emulation. This allows a lighter emulation environment and more control over the emulated program. It also opens the possibility to model simpler processor chips that do not run operating systems but interact with the program on a lower level.

The models currently emulated are compiled for a standard x86 Linux or Windows distributions (64-bit mode). This stems from the necessity to also run the model programs in a non-emulator environment for lighter usage in larger scale simulations where the time aspect of the program execution is not important. In this case, the models can be loaded as a Windows or Linux library directly into the simulator.

In order to emulate such programs, the emulator must carry out the roles of an operating system, i.e., loading the program into the computer's memory, linking it to system functionalities, and providing a way to locate and call the functions of the program. This is embodied in the *OS* component of the *Computer* model, which is currently instantiated with a Linux or Windows version of the operating system emulation.

1) Program loading: The first step in loading a program into memory is parsing it. The format of the program files is the *Portable Executable (PE)* format for Windows and the *Executable and Linkable Format (ELF)* for Linux (and Unix-based systems in general). Both formats are mainly used for executables and libraries. They start with a header to identify the format, the target system, and mode. These headers point to the different structures contained in the file and identify various tables. The structure of a Linux program is shown in Fig. 2. The PE format has some differences but in this context, the functionality is similar to ELF files.

The parsing of Linux programs is performed manually using the structures and values defined in the Linux source code (*elf.h, elfcode.h, elf-em.h, . . .*). The created ELF parser reads a file following the format and fills the structures with the file's data. This is done by placing pointers of the structures' types at the right positions on the original file data. The pointers to the start of tables can be used directly to access the table's content through indices. Parsing of Windows programs is performed with the help of the *pe-parse* library.

The names of different elements in the program files such as section names or symbol names are located in *string tables*. This common mechanism puts all the strings of a program in a block separated by the string termination character ('\0'). The places referring to names or strings in the file then just give the start position of the string in the string table. The location of the string tables is given in the headers of the files.

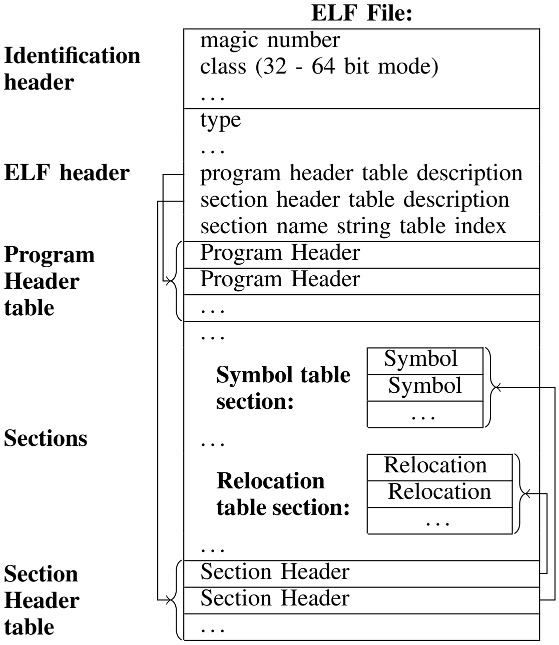

Fig. 2. Overview of the ELF file structure.

The formats specify a list of chunks in the program file that have to be loaded into the memory of the computer at correct virtual addresses. In ELF files, these are the *Program Header* entries which specify a part of the file (that can contain multiple sections). In PE files, there are just sections in the file, but flags specify if it has to be loaded into memory at runtime. These chunks contain, e.g., the program code, the program data, and the jump tables for external functions. Using the *Memory* component of the emulator, a *MemorySection* is allocated for every section of the program. The correct read, write and execution permissions are set depending on the descriptors of the program file.

The next step is reading the symbol tables and registering the symbols contained in the file. The symbols representing public objects and functions are called *Exports* in the PE format and are in a special table. The symbols in the ELF file are not sorted in this way but have additional descriptors giving information about linkage (from which the functions of the program can be deduced). These are registered in the symbol table of the emulator (hashmap with the name as the key and the virtual address as value) and later used to discover the interface of the EMA model.

2) System functionalities: The program also has to be able to make calls to the operating system. These function symbols are not resolved at compile time so they have to be set by the operating system or in the present case by the emulation of the operating system. The places where the program needs those symbols are listed as *Relocations* in ELF files. In PE files these are in a list of *Imports*.

An ELF *Relocation* or a PE *Import* specifies a location where the address of a system call has to be written in the virtual memory. This location is an entry in a "jump table" (or branch table). Programs use the `call` instruction to make function calls. It is responsible for pushing the return address to the stack, but it can not jump to a variable location. This is accomplished by a special `jmp` instruction placed at the target location of the `call` instruction. This variant of the jump instruction is set to jump to the location stored at a given address. This address is the address of the jump table entry and is coded into the instruction. The combination of these two instructions is a standard way of making calls to functions not resolved at compile time.

Those system functions are not inside the emulator. For this, an escape system is implemented where the address given to the program points to a special section in the virtual space. Using the monitoring hook of the Unicorn engine, the emulator checks if the next instruction happens in this section. If so it catches the call and looks up the registered external system functions, which are implemented outside the emulation. These external functions can use the different components of the *Computer* to emulate the actions of the called function.

To return to the program execution, the effects of the `ret` instruction have to be emulated. This is performed by setting the instruction pointer register to the last value on the stack. This value is popped from the stack using the *VirtualStack* component.

When the program tries to call a function that has not been implemented in the emulator, it is still safely caught and the emulator tries to return 0 to the program. It also notifies the user, so that critical missing functions can be identified and implemented. The *OS* component implementations are responsible for registering the available system functions so that they can be linked when loading the program. Implemented functions include *malloc*, *sqrt*, *sin*, *cos*, *acos*, *exp*, *memcpy*, *strlen*, *strncmp*, etc. The address of a special *exit* system function is always placed at the top of the program stack before calling program functions so that the function return can be caught and the emulation exited.

Since the programs used here are compiled as libraries, the last step to load the program is to call the initialization function of the library. This is the *_init* function in Linux and the entry point defined in the PE header for Windows.

3) Function calling: The last role of the *OS* implementation is defining the function calling standard used under the specific system. This is done through the implementation of a *FunctionCalling* interface. This interface allows the calling of functions without knowing how the arguments are passed to the emulation or how the return value is read.

Both Linux and Windows operating systems and compilers use a variant of the *FastCall* standard. It works by passing the arguments using different registers depending on their type and order. Under Windows, the arguments of a function call are passed left to right in the RCX, RDX, R8, R9, R...registers for all integer or pointer types. Under Linux, they are passed inside the RDI, RSI, RDX and RCX registers. The return value is passed through the RAX register.

For floating-point values, the arguments and return value are passed in the XMM*n* registers. The number *n* goes from 0 to 15 for modern processors, and they are used in this order for arguments left to right. In the case of a floating-point return value, it is always placed in the XMM0 register.

E. Execution time models

The main goal of the emulator is to have a time model for programs running inside the simulated world. A *ComputerTime* component is responsible for collecting the timing evaluations from the different emulation components. It is configured with the CPU and memory clock frequencies (in Hertz) and provides conversion function from CPU or memory cycle counts to time. This component manages two time-precisions: picoseconds and milliseconds. The picosecond precision is needed because the duration of CPU cycles are in the nanosecond range (ex: 1s / 1 GHz = 1ns). To keep this cycle-time precise when using integers, it is computed in picoseconds. The component proposes three methods to add time to its internal clock: *add_cpu_cycles(cycle_count)*, *add_memory_cycles(cycle_count)* and *add_pico_time(time)*. The first two use the defined CPU and memory frequency to convert the cycle count to time. In all cases, it converts the picosecond time to microseconds, which is relevant for the simulation and keeps the modulo for precision.

The time evaluation is possible through the registering of *monitoring hooks* in the Unicorn engine. The engine will then call these hooks while emulating the program. There are hooks for instructions, memory, and errors. The instruction hooks get called before every single instruction execution and the memory hooks get called for every memory read or write operation caused by an instruction. The error hooks can be used to gather information on what caused an error inside the emulation.

1) Processor time: The evaluation of the time used by instructions directly (not counting the memory access time) is performed by the *CodeDecoder* component. It is called by the monitoring hook for every instruction and reads the bytes of the instruction from the engine's memory. It then uses the *ZyDis* library, which can decode x86 (and AMD64) instructions, to get an enumeration value for every type of instruction. This value is used directly to lookup a CPU cycle count in a time table. This cycle count is cycle added to the *ComputerTime* component.

The time table currently contains values from a benchmark of an *Intel Skylake* processor. This table can be changed to represent other CPUs. This model does not currently take into account the instruction context. Some of the tables found specify different tick counts depending on the size or type of data the instruction works on. It can be different if the data is constant and coded into the instruction, or if the register used is bigger or a floating point register.

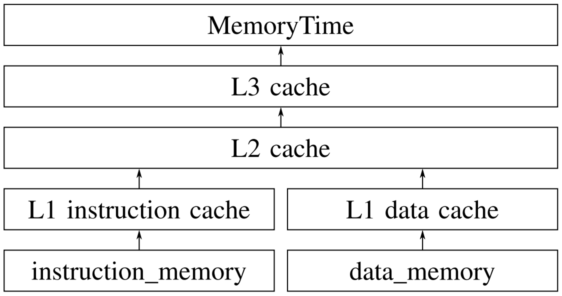

Fig. 3. Example of a cache configuration with multiple layers.

2) Memory time: Another big factor for code execution time is the interaction time with the computer's Random-Access Memory (RAM). This happens when instructions themselves are read by the control unit of the CPU and when instructions read or write data from or to memory. These cases are managed by the *MemoryModel* component. It is called by the monitoring hook for every memory access and for every instruction execution (because the Unicorn engine does not call the memory hooks for the reading of instructions).

The time it takes to write and read the main memory is encapsulated in a *MemoryTime* component. This component is configured with the number of memory cycles necessary for reading and writing 8 bytes of memory (a memory "block").

With current CPU technology, memory times are an order of magnitude larger than CPU times. This is why caching is standard. It allows data to be stored in a smaller but faster memory so that the CPU does not have to wait for data to arrive from main memory. Since caching has a large influence on computing times, the *MemoryModel* contains a configurable way of setting up cache layers.

Instead of directly asking the *MemoryTime* component for the necessary time, the *MemoryModel* uses two *MemoryAccessInterfaces*. One for data memory access (*data_memory*) and one for instruction reading (*instruction_memory*). The *MemoryAccessInterface* is implemented by the *MemoryTime* component and also by any optional cache model. The two interfaces to data and instruction memory can then represent a stack of cache layers finishing with the *MemoryTime* layer. This is shown with the example configuration in Fig. 3.

Each layer is queried for the time it takes to read or write a certain address. The cache layers can then evaluate if the address represents a cache hit or miss. In the latter case, it can add the time from the next layer, and so on until eventually hitting the *MemoryTime* layer.

The current cache implementation has a configurable size and its reading and writing times are expressed in CPU cycle counts. It implements the FIFO *Replacement Policy*. A Replacement Policy is the algorithm used to choose which entries are replaced in the cache when it is full and a new entry has to be added. Other policies such as the *Least Recently Used* policy or more complex heuristic based policies could be implemented and placed in the *MemoryAccessInterface* stack.

3) Operating system time: Currently, calls to system functions are not covered by the time evaluation since they are performed outside the emulation environment. Therefore an approximation time is currently added by every external system call. These time models provide an approximation for the execution time of software that can be used in the overarching simulation (R2). Alternatively, the operating system functions would have to be integrated into the emulation through an operating system image. This would require relocations, inter-program linking and proper setup of such an image.

F. EmbeddedMontiArc and MontiSim Integration

The MontiSim simulator uses the created emulator to simulate the software of its autonomous vehicles. The emulator is given an EMA model to load and the desired configuration for the computer time models. The simulator can the set and read the ports of the model depending on the internal vehicle communications. The execution time can then be read from the emulator in order to delay the emulated software accordingly.

The software components of a simulated vehicle are set up in an E/E infrastructure simulation. This E/E simulation works with the *Discrete-Event* paradigm, where processes are not updated on a tick basis but their effects and durations are saved in events. A *Discrete-Event Simulator* then processes events ordered by their finish time.

Since the emulated EMA models work on a cycle basis, their execution can be performed in one block. The evaluated execution time is then used to create a discrete event that will trigger the outputs of the model in the future of the simulation. This discrete-event system allows the modeling of different E/E components inside the vehicle, such as different computers, different buses, the sensors and actuators of the vehicle, etc. An example of the generation of a discrete event by the Hardware Emulator is shown in Fig. 4. It also shows the interaction between the tick-based part of the simulator (physics simulation) and the discrete-event system. For a given simulation *tick*, the simulator generates events that represent new physical data (such as the sensor data). The discrete-event simulator then processes events until the next tick time, on which this cycle continues.

In order to dynamically integrate EMA models in the simulator, an *Adapter* generator was created that can use the EMA language parser and generator to generate its own set of functions depending on the model. The generated *Adapter* contains a set of predefined functions that list the number, name, and type of input and output ports of the EMA model. It is compiled as a layer over the generated model code. This allows the simulator to dynamically discover the ports of the model. From their name and type, the simulator can deduce the name and signature of the generated function that is used to interact with the ports of the model. This dynamic approach allows the model to define the communication channels it uses inside the simulation dynamically. All the required functions (including the port discovery functions) of the model can be looked up in the symbol hashmap of the emulator that is filled by the program loaders.

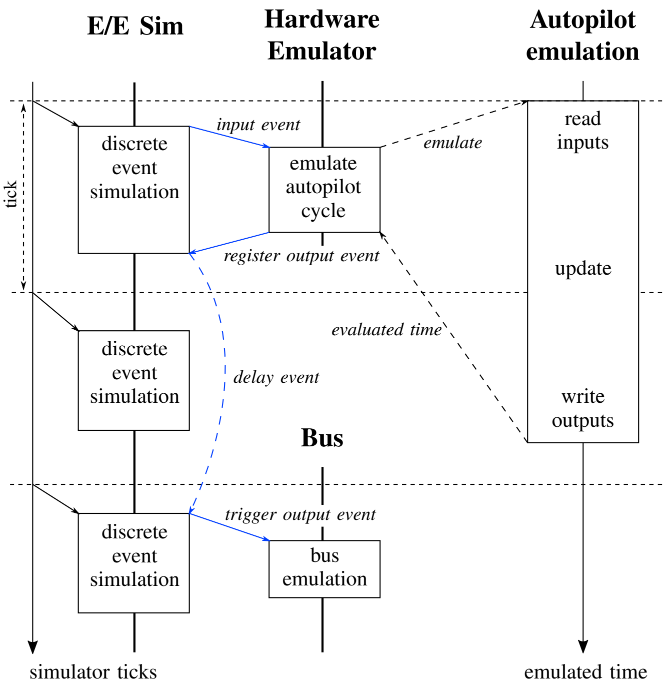

Fig. 4. Discrete-event interaction of the Hardware Emulator. Calls in blue are related to *Discrete Events*.

Name	CPU frequency	Memory frequency	Data and instruction L1 cache	L2 cache
Basic	10 MHz	1 MHz	128,1,2	1024,10,15
Slow CPU	1.5 MHz	150 KHz	128,1,2	1024,10,15
No cache	10 MHz	1 MHz	—	—

TABLE I
HARDWARE CONFIGURATIONS.

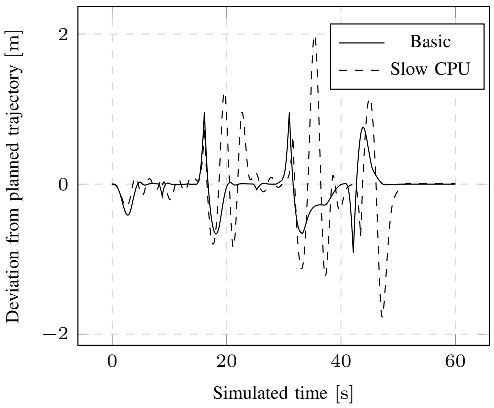

Fig. 6. Deviation from planned trajectory for different hardware speeds. (Positive deviation is left of the trajectory.)

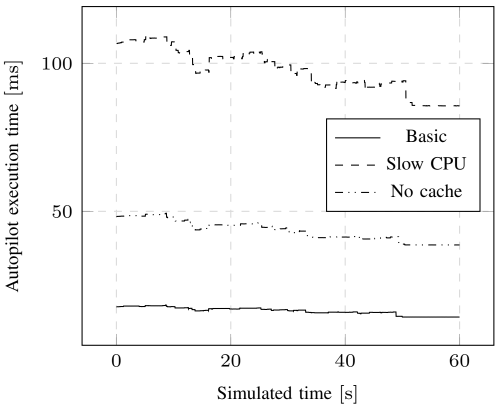

Fig. 5. Execution times of an autopilot cycle for different hardware configurations.

V. Evaluation

The following evaluation will show the simulation results for a series of virtual computer hardware configurations. The simulation setup is using a part of the *Aachen* city exported from *OpenStreetMap*. The simulated vehicle uses a mass point physical simulation and is set up with a simple autopilot (the emulated software). The different hardware configurations tested are shown in Table I. The tuple of numbers for the cache settings represent the cache's size, read and write time (in this order). The read and write times are expressed in CPU cycle count. The evaluation of the single instructions takes values from a table of an *Intel Skylake* benchmark.

One of the metrics exported from the simulation is shown in Fig. 5. It is the evaluated software execution time for one cycle of the autopilot: reading the inputs, executing the logic and writing the outputs. The other exported metric is the deviation of the car to the planned trajectory (shown in Fig. 6). This represents how well the autopilot can control the car and follow its target trajectory. The *planned trajectory* is a set of line segments that go from the location of the car to its target coordinates. The deviation is then computed by taking the orthogonal distance to the closest trajectory segment. The sign of the deviation represents the side of the trajectory on which the vehicle is.

Fig. 5 shows the influence of the hardware configuration on the evaluated time. The difference between the *Basic* and *Slow CPU* simulations is due to slower CPU and memory frequencies. The difference between *Basic* and *No cache* is only due to the presence or absence of a cache model for the CPU. Fig. 6 shows the influence of hardware to slow for the software on the controlling capabilities of the autopilot software. In the *Slow CPU* simulation, the autopilot has bigger difficulties to follow the planned trajectory and frequently falls into an oscillation pattern around the desired trajectory.

VI. Related Work

There are basically two different approaches to simulating an embedded system: One can either extend an existing general purpose simulator or create a new simulator from scratch. In case of cooperating embedded systems, e.g., in Internet of Things (IoT) or cooperative driving applications, extending the discrete-event network simulator ns-3 [9] is a popular option because of its large number of network and mobility models. However, this advantage comes at the cost of needing to provide a simulator-specific implementation.

iTETRIS [10] is an Intelligent Transportation System (ITS) simulator that combines ns-3 with the traffic simulator SUMO [11]. A disadvantage of this framework is, however, that ITS applications have to be implemented using the ITS simulator of iTETRIS. Therefore, a second implementation would be required if the application should be deployed on real hardware. The Direct Code Execution (DCE) [12] extension of ns-3 allows simulating the execution of binaries compiled for Linux. Similar to our approach, DCE replaces function calls that depend on the host machine by simulator-specific code. CoWS [1] uses DCE to simulate code written for the Wireless Open Access Research Platform (WARP) [13]. CoWS replaces hardware-specific function calls of WARP devices, such as setting the transmission frequency, by ns-3 compliant implementations using the mechanisms offered by DCE. The fact that CoWS uses almost the same code that would be executed on the target platform allows the developer to analyze the code using standard debugging tools like `gdb` that might not be available on the target platform. Disadvantages of this method are that the executable has to be linked specifically for ns-3 and that it requires a small number of modifications to the code to return the control to the simulator during endless loops. A major difference to our approach is that our approach allows using unmodified libraries in the simulator by using an emulator instead of simulating the behavior of the hardware. Alternatively, it is also possible to use an architecture description language to generate code targeted at ns-3 [14]. However, as the generated code relies on ns-3 specific interfaces, it is not executable on real hardware without adapting the code generator.

Instruction Set Simulations (ISSs) allow simulating embedded systems at the level of single CPU instructions. However, this creates a considerable overhead [1]. In contrast, EMA focuses on modeling the software architecture of embedded systems. To nevertheless provide results close to the results achieved on real hardware, we allow specifying certain parameters of the hardware that allow us to abstract from specific hardware components while still achieving comparable results.

[15] describes how to create software architectures that combine models with different Models of Computation (MoCs). This is achieved by keeping the MoC consistent on each hierarchy level. Different hierarchy levels, i.e., nested components, may, however, use different MoCs. Similarly, we separate the autopilot emulation from the discrete-event simulation and only exchange the results of the computation with the simulation.

VII. Conclusion

In this paper we proposed a reproducible hardware emulation approach making the simulation of model execution more realistic. The approach analyzes a compiled model and predicts its runtime for the desired architecture based on the generated assembler code. It is based on the Unicorn emulation engine and considers the execution duration of the instructions as well as memory access and caching.

We integrated the emulator into a vehicle simulator to simulate the delays of an autonomous driving controller. In a trajectory following experiment, we showed how different hardware configurations affect the trajectory driven by the simulated vehicle. Obviously, a model execution simulation is indispensable for the assessment and validation of embedded system models. Future work comprises experimentation and evaluation with more complex E/E infrastructures and comparisons with real hardware.

References

[1] Martin Serror, Jörg Christian Kirchhof, Mirko Stoffers, Klaus Wehrle, and James Gross. Code-transparent Discrete Event Simulation for Time-accurate Wireless Prototyping. In *Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, pages 161–172, New York, NY, USA, 2017. ACM.

[2] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[3] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[4] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[5] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[6] Mordechai Haklay and Patrick Weber. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[7] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn-the ultimate cpu emulator, 2015.

[8] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[9] The ns-3 Consortium. ns-3 Discrete Event Network Simulator. [Online] https://www.nsnam.org.

[10] Jérôme Härri, Pasquale Cataldi, Daniel Krajzewicz, Robbin J. Blokpoel, Yoann Lopez, Jeremie Leguay, Christian Bonnet, and Laura Bieker. Modeling and Simulating ITS Applications with iTETRIS. In *Proceedings of the 6th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*, PM2HW2N '11, pages 33–40, New York, NY, USA, 2011. ACM.

[11] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent Development and Applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012.

[12] Mathieu Lacage. *Experimentation Tools for Networking Research*. PhD thesis, Université de Nice-Sophia Antipolis, 2010.

[13] Rice University and Mango Communications. The WARP Project. [Online] https://www.warpproject.org/trac.

[14] Mihal Brumbulli and Emmanuel Gaudin. Towards Model-Driven Simulation of the Internet of Things. In Michel-Alexandre Cardin, Saik Hay Fong, Daniel Krob, Pao Chuen Lui, and Yang How Tan, editors, *Complex Systems Design & Management Asia*, pages 17–29, Cham, 2016. Springer International Publishing.

[15] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming Heterogeneity—The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.