Virtual Prototyping Evaluation Framework for Automotive Embedded Systems Engineering

Sebastian Reiter[1],Andreas Burger[1],Alexander Viehl[1],
Oliver Bringmann[1,2],Wolfgang Rosenstiel[1,2]
[1] FZI Forschungszentrum Informatik, Haid-und-Neu-Str. 10-14, D-76131 Karlsruhe, Germany,
[sreiter, aburger, viehl]@fzi.de
[2] Universität Tübingen, Sand 13, D-72076 Tübingen, Germany
[bringman, rosenstiel]@informatik.uni-tuebingen.de

ABSTRACT

This paper presents an analysis framework based on virtual prototyping to support the comprehensive evaluation of distributed, network based automotive applications. The framework enables functional and timing verification, performance and reliability analysis while reducing the evaluation complexity. Additionally the framework supports design space exploration of the overall system, considering target hardware/software and the system environment. The presented approach closes the evaluation gap between the initial design and the final system integration test. During the whole design process the analysis supports the designers in reaching efficient design decisions. The applicability of the proposed framework is demonstrated by representative automotive use cases. Highlighted are benefits like the integration of existing software prototypes or the automation capability. The performance comparison with a widely used network simulation tool shows the competitiveness of the presented analysis framework.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems; I.6.5 [**Simulation and Modeling**]: Model Development; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Performance

Keywords

Virtual prototyping, system simulation, performance evaluation, automotive, SystemC

1. INTRODUCTION

In the last two decades the amount and complexity of software within vehicles has increased exponentially. In current premium cars more than 70 electronic control units (ECUs) are integrated in a networked environment, leading to an increasing variety of design alternatives. During the design process these alternatives have to be assessed to reach efficient design decisions. Besides major decisions like a suitable communication technology or topology, a wide range of parameterization alternatives exists that drastically influence the final system behavior. Each of the decisions has to be verified that they still fulfill the functional, reliability and performance requirements. With today's demanding requirements, the available resources have to be used efficiently; a general resource overestimation is often not feasible. Design decisions are nowadays based mainly on spreadsheet information, experiences with previous products or analyses of reference system designs. The influence of design decisions onto the actual system are verified in late design phases when physical prototypes are available, making it almost impossible or very cost-intensive to change the overall system design. Another aspect is to determine the effects of design changes, not only the actual system has to be taken into account, but also the interactions with other system components of the networked environment. Therefore a detailed, overall system evaluation along the design process is needed, which closes the evaluation gap between the initial design and the final system integration test. Figure 1 highlights the classic V-Model enhanced with the proposed approach.

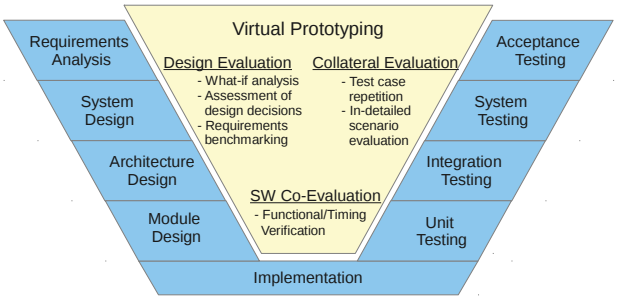

Figure 1: Virtual Prototyping fills the evaluation gap between the design and testing phase

The presented evaluation framework uses virtual prototypes to evaluate the system under development along the design process. A virtual prototype is a behavioral model of the actual system. It enables what-if analyses before committing to one design alternative. The presented approach enables the usage of abstract models in early design phases to evaluate and verify coarse system aspects. With the progress of the design process the model is refined, getting more accurate

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254625

in mirroring the behavior of the physical prototype. Each refinement step is sophisticated enough to execute the required analyses of the current design phase. When software prototypes are available they can be integrated and tested in combination with the simulated system parts. In late phases when physical prototypes are available, the virtual prototype adds still benefits to the design process. Features like halting the complete system and monitoring each internal state, provides an advantage over system debugging with physical prototypes. The provided deterministic behavior allows a repeated simulation of test scenarios executed with physical prototypes. Compared to physical prototypes and HIL simulations the evaluation overhead is generally smaller. Co-simulation approaches increase the capability of the simulation platform, e.g., by providing a target code interpretation or the simulation of the physical environment. Implementing this approach the following challenges emerge.

- Support for a variety of system alternatives and system refinement
- Reusing applications in different system scenarios
- Integration of existing software prototypes
- Designing and configuration the simulation with manageable effort

A solution of these challenged is presented in the following paper. Section 3 highlights the elaborated evaluation framework and motivates the choice of SystemC [6]. In Section 4 approaches to support a variety of system alternatives are shown. Section 5 presents the configuration approach, which enables an efficient handling of the system variant space. Section 6 shows the resulting design flow and a graphical environment supporting the user. Section 7 highlights concepts for integrating existing software prototypes. Use case evaluations are shown in Section 8.

2. RELATED WORK

There exist different approaches using virtual prototypes to evaluate a variety of system aspects. In [8] virtual prototyping is used to evaluate the interaction of two communication technologies, Bluetooth and WiMedia Ultra Wide Band. A FlexRay communication controller model at the Register Transfer Level (RTL) is used in [7] to support intellectual property (IP) development. Other simulation-based approaches are presented in [10, 14, 2]. All these approaches focus on dedicated system parts and particular evaluation goals, resulting in an optimized virtual prototype for dedicated analyses. They do not target a generic, reusable evaluation infrastructure, neglecting the models extensibility and reusability. The overall system and the goal to iteratively reach a comprehensive simulation platform is not the focus these approaches. [4] presents extensions to SystemC, supporting the simulation of embedded systems and the surrounding network environment. Supporting a variety of system alternatives or the efficient simulation configuration is not regarded. Specific extensions could be integrated into our evaluation framework, but concepts from transaction-level modeling (TLM) are currently sufficient.
Besides the research approaches, a number of commercial solutions such as [19, 20, 16] exist, allowing a component-based assembly of virtual prototypes. These tools assemble detailed IP components, Instruction Set Simulators (ISS) and interconnection networks to simulate a system. Most of these tools don't allow a very abstract architectural exploration, because the provided IP blocks are already technology dependent and lacking the capability of model refinement. Another aspect is the integration of application software. In most tools the software has to be cross-compiled to run on an ISS, meaning a flexible hardware/software partitioning is diminished.
Regarding the communication part there exist a range of network simulation tools, such as [15, 17, 18]. These tools focus primarily on building networks for simulation. It would be possible to extend the discrete simulation to cover on-chip resources, e.g., shared processing resources. SystemC on the other hand is originating from the hardware description area and is evolved to a system-level modeling language. In the presented approach SystemC is given preference to existing network simulation tools.
Other approaches suggest analytic frameworks to evaluate different isolated system aspects. In [3, 5, 9] timing and predictability analyses of a FlexRay bus system are presented. These approaches are based on analytic methods like Real-Time Calculus, ILP or mathematical models. All approaches have in common that the system behavior has to be transferred to the analysis model, e.g., *Arrival* and *Service Curves* have to be derived for the Real-Time Calculus approach. A mathematical model of the access times has to be specified for [5]. The applicability of these approaches is therefore limited to evaluate isolated system aspects.

Most of the mentioned approaches focused on the evaluation of dedicated system aspects or platform dependent code, while neglecting the overall system. The analyzed systems are modeled at a fixed level of abstraction, the capability of model refinement and system variation isn't considered. Reusing already specified system parts across different evaluation scenarios isn't regarded.

3. EVALUATION FRAMEWORK

The presented evaluation framework is based on a set of behavioral models, called virtual prototype. Virtual prototyping enables the evaluation of hardware/software systems without the need of physical prototypes. With the help of a software-based simulation kernel, the required system parts are simulated. The event-driven simulation language SystemC enables the creation of virtual prototypes that can be used to describe the complete system or to interact with already developed software implementations. With these mechanisms the functional and timing behavior of combined hardware/software systems can be simulated. Language features like modules or ports encourage a hierarchical, composable design. Based on the fact that the actual software and the hardware are described by a software-based simulation the hardware/software partitioning can be flexibly varied and explored. Both the timing of the software and the hardware can be modeled and refined during the design process. Approaches like [13] allow estimating the execution time of software components according to the target architecture and annotate them to the source code. This enables the timed software simulation from a very abstract level to a detailed architecture depend level without the need to simulate the execution platform. With the TLM capability, SystemC offers an extension towards system-level modeling. TLM2.0 enables the modeling of communication apart from specific hardware implementations across a variety of abstraction levels, such as cycle-accurate, approximately timed, loosely-timed or untimed. The inherent concept of unified interfaces and sockets increase the interoperability of modules. The availability of different co-simulation approaches, for example, the integration in a driving simulation, similar to [21], allowing the user to dynamically interact with the virtual prototype, supports the choice of SystemC. Another aspect is that an increasing number of IPs is already provided with additional SystemC simulation models.

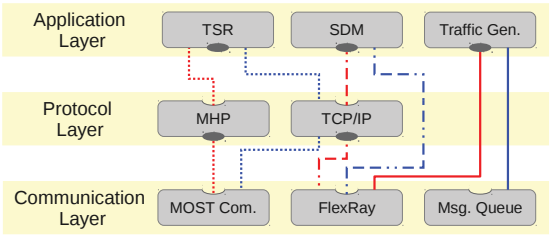

Figure 2: Sketch of the three layered approach

One general advantage of the virtual prototyping is that only the necessary aspects of the evaluated behavior have to be implemented. Resulting in an evaluation oriented implementation; this is generally less complex than the actual system implementation. As example, the developed Media Oriented System Transport (MOST) model provides implementations of the asynchronous, synchronous, control and the isochronous data exchange. It simulates a MOST frame, which circles the MOST ring and is clocked with the typical MOST frequencies by a timing master. A real MOST network would forward this frame in a bit-by-bit manner, via a physical layer, most likely plastic optical fibers (POF). This aspect is neglected in the implementation and the complete frame is forwarded on a physical bus segment with a fixed delay. Introducing a bit-by-bit simulation wouldn't benefit the current evaluations and increase the simulation runtime and implementation complexity significantly.
The evaluation of automotive applications in combination with different communication systems is the main focus of the presented analysis framework. The current implementation covers network technologies like MOST, FlexRay, a half-duplex Ethernet and abstract TLM-2.0 based communication models. Different communication protocols are implemented on top of this communication layer. We focus on a TCP/IP stack and the MOST High Protocol (MHP) in this paper. In the scope of previous industrial projects different applications are integrated. They range from dedicated industrial use cases, such as a traffic sign recognition (TSR) or a stereo depth map (SDM) application to a set of generic traffic generators. The traffic generators can be used if only the communication characteristics of the application are given. This is important if the application couldn't be disclosed because of confidentiality reasons or the effort to port the application onto the virtual prototype isn't worthwhile, for example, if only the interference of an additional application onto a shared bus should be taken into account.

Resulting challenges: To foster the evaluation of various systems, different parts of the simulation have to be reused. It should be possible to use the same application with different communication technologies, without changing the application. Parameters of the system, such as the communication bandwidth, the internal buffer sizes or the host system performance characteristic should be adjustable without redesigning the simulation. If parts of the system have to be changed, e.g., a different communication protocol version or a different level of abstraction, replacement should be limited to these parts. The approaches addressing these topics are elaborated in Section 4.

4. VIRTUAL PROTOTYPE STRUCTURE

The developed concepts to foster a configurable, reusable simulation platform are highlighted in this section. These cover the layered approach with standardized interfaces, the used addressing approach and the parameterizable modular sub-structure.

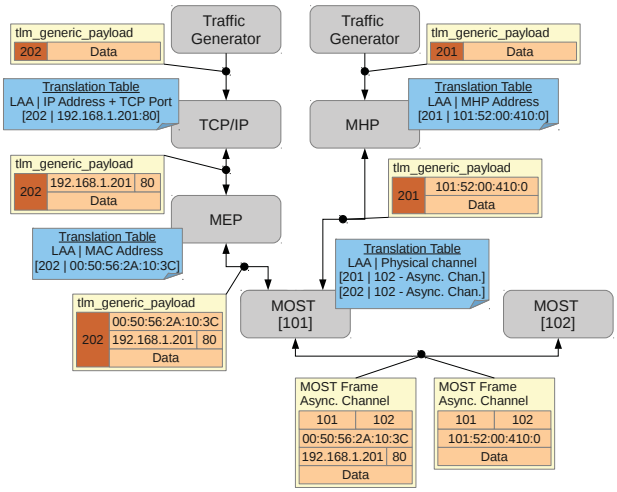

Figure 3: LAAs within the TCP/IP, MHP scenario

4.1 Three Layered Approach

The virtual prototype uses a three layered approach, consisting of the application, protocol and communication layer, as shown in Figure 2. The modules of the different layers are interconnected by TLM-2.0 interfaces. Because modules communicating between layers are using the same standardized interface, the exchange of subjacent modules is enabled. The application layer contains data sink or data source modules. All require a TLM-2.0 interface of a subjacent communication module to transmit and receive data, depicted as filled ellipse in Figure 2. The middle layer offers protocol mechanism. Modules of this layer provide a TLM-2.0 interface to the upper layer and request a TLM-2.0 interface from the lower layer. They process and forward incoming data, most likely by adding protocol specific information to the payload, e.g., a TCP/IP header and trailer. Nevertheless this layer can act independently from the application layer, in case of retransmission or segmentation. The lowest layer implements the actual communication technology like MOST or FlexRay. These modules offer a TLM-2.0 interface and handle the data exchange between different devices.
With this approach it is possible to map an application onto different communication technologies, because all provide a standardized interface. Further it is possible to concatenate various protocol implementations, because they provide and require the same interfaces. Figure 2 highlights different alternating communication paths. The same TSR module uses either the MHP or the TCP module from the protocol layer. The SDM application module is mapped either to the protocol layer or directly to the communication layer. The traffic generator is used either with the FlexRay bus or with abstract TLM message queues, illustrating the exchangeability of communication modules.

4.2 Logical Application Address Approach

When changing the communication technology, the heterogeneous addressing schemes cause a problem. The TLM-2.0 interfaces use a generic payload structure to exchange data. This structure allows specifying the data length, the actual data pointer and the target address with a length of eight bytes. The problem is that different protocols and communication layers are using different addressing formats, e.g., TCP/IP uses IP addresses and port numbers, MHP is using a vector of FunctionBlock, InstanceID, FunctionID and OperationType and the FlexRay bus uses slot IDs. Using these

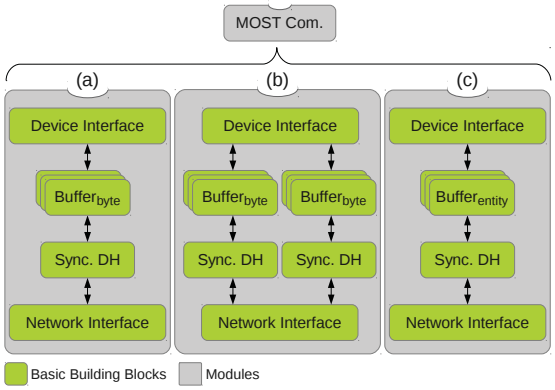

Figure 4: Basic building block assembly

physical addresses within the application would eliminate the flexibility to use one application with different communication layers. To regain this flexibility a *logical application address* (LAA) is introduced. The LAA identifies a communication association, meaning if two applications send data to the same target devices different LAAs are used. Every time a module needs a *physical address* this module provides a translation table to decode a LAA into a *physical address* or vice versa. The according *physical address* is only used in the communication module or encapsulated in the payload data. Meaning all modules of the application and protocol layer, which are using the TLM-2.0 interface for data exchange, are using a *logical application address*. Within the communication layer only *physical addresses* are present. If two communication associations have to be mapped to the same channel, both *logical application addresses* are mapped to the same *physical address*. In Figure 3 the different translation tables of a TCP/IP, MHP scenario are presented. Both applications are using a different LAA to identify the communication destination. If the involved communication modules need a physical address, a translation table is provided. For example the TCP/IP module needs the *physical address* to generate a TCP/IP header. After the header generation and the data encapsulation, the frame is forwarded with a *logical application address*. At the end both messages are mapped to the same physical channel, with the same physical target address.

4.3 Modular Sub Structure Approach

The three layered approach combined with the LAA offers a first degree of flexibility. It is possible to map different applications onto a variety of protocols and communication mechanisms. For gaining the required flexibility to evaluate a variety of system alternatives and variants, the functionality of the different parts is further divided into sub modules. The different communication, protocol and application modules are assembled by connecting the required sub modules, here called basic building blocks. In the following this approach is highlighted with the help of a MOST model.
The functional description of a MOST communication module is aggregated from a set of 17 blocks, each realizing basic tasks. For each MOST channel type, e.g., a dedicated building block exists that extracts the data from the frame and stores it into a buffer or vice versa. By assembling a MOST module from these basic building blocks, it is possible to create different amounts of channels, such as multiple synchronous channels, compare Figure 4 (a) and (b). As a result, different virtual prototypes can be realized by just changing the block assembly.

Another aspect is that the functionality of single blocks can be implemented in different ways. A MOST device, e.g., contains a buffer block that stores receive or transmit data. In the current model two basic block implementations with different internal memory representations and allocation algorithms are present. Both implementations are using a common interface and are therefore exchangeable. The behavior of the device changes when these blocks are exchanged in the assembly, compare Figure 4 (a) and (c).
With the functional partitioning it is possible to describe the same modules at different levels of abstraction. If the functional and timing behavior of a basic building block is abstracted because a more detailed modeling wouldn't benefit the evaluation goal, a more abstract block is used to increase the simulation performance. If another evaluation needs a more detailed functional or timing behavior, the basic block is substituted with a more accurate implementation.
Besides the flexibility of the basic building block assembly, most blocks provide a set of parameters that allow adjusting the behavior. The 17 basic building blocks of a MOST module offer a total set of 48 configurable parameters. Most of them are used to personalize the devices, e.g., the device address or to provide assembly dependent information such as the LAA translation tables. Other parameters influence the behavior of the system and allow evaluating a greater set of alternatives, e.g., buffer sizes, interrupt rates or the provided bandwidth. With this set of parameterizable basic building blocks a variety of systems can be specified.

Resulting challenges: With such a degree of flexibility, it is not suitable to assemble and configure the desired system in source code. Each time a parameter changes or a module is swapped, the complete system would have to be recompiled. Especially when large sets of simulation runs have to be executed with small changes in the parameterization, e.g., used to find the best system parameterization, the total simulation time would increase drastically. Additionally, writing a *Top Module* and passing all parameters with the help of constructors can be a very time consuming task. The following Section 5 presents the used system configuration approach to solve these challenges.

5. SYSTEM CONFIGURATION

The evaluation framework uses an xml based configuration approach. The composition and the parameterization of the basic building blocks are given as separate configuration file. The information of this file is used during execution to assemble and configure the system. With this approach it is possible to simulate different system alternatives without

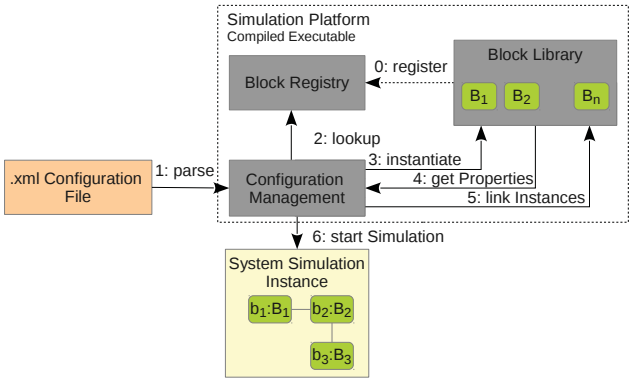

Figure 5: Virtual prototype configuration approach

recompiling the system, by just changing the configuration file. In Figure 5 a sketch of the process is shown. The basic blocks, presented in the previous section, are summarized in the *Block Library*. Each single block of the *Block Library* is registered in a central *Block Registry*. The *Block Registry* is a lookup table where each basic building block is associated with a unique ID and a function to instantiate the block, similar to a factory method. The second key element is the *Configuration Management*. The task of this class is to parse a configuration file and provide the contained information to the simulation platform. After the xml file is parsed, all contained blocks are instantiated. Therefore each block ID is looked up in the *Block Registry* followed by the factory method call, returning the created object. The *Configuration Management* stores the created object reference by an object ID specified in the xml file. An iterator that allows stepping through the configuration parameters, associated with the current block, is passed with the factory method. Each parameter consists of a unique ID, a data type and a value which are used to initialize the member variables. It is possible to cascade parameters and build complex data structures. The responsibility to interpret the configuration parameters and assign them to the correct variables lies with the basic building block. The *Configuration Management* only provides the unified access to the information. After all required blocks are instantiated, the blocks are interconnected. To prevent the need for a dependency analysis of the blocks and to allow circular dependencies between blocks, the instantiation and the linkage are done in two separate steps. The *Configuration Management* iterates through the links of a block, calls the linker method and passes a reference of the target block. The block instance uses the information to establish the connection. The object is responsible to correctly cast the unified reference and establish the connection in a correct manner. In the current simulation platform normal object pointers, TLM sockets and SystemC ports are used. By typecasting the generic block reference, configuration errors within the xml file are revealed and the simulation is terminated. After these two steps are executed for each basic building block, the *Configuration Management* contains interconnected instances of blocks assembled to a complete system. With this system simulation instance the simulation is executed.

Changing a parameter value only requires the modification of the xml file. The simulation can be re-executed without recompilation. Additionally it is possible to exchange modules in the xml file.

There exits a variety of xml binding tools for various programming languages, which enables the easy creation of powerful test generators. The xml instance specification is used to automatically generate code that allows parsing and serializing the xml files. In different analysis scenarios Python scripts are used to automatically generate a huge amount of xml configuration files.

Resulting challenges: Each basic building block contains a similar factory and linker method. This repeated functionality is most suited for automated code generation. The information, parameters and links, specified for this step, can be additionally used to generate the xml configuration files. The graphical tool and the code generation are presented in Section 6.

6. GRAPHICAL TOOL SUPPORT

To reduce the overhead of designing the virtual prototype, different code generation steps are used. As explained in the previous section the virtual prototype contains repeated code templates, which are most suited for code generation. The

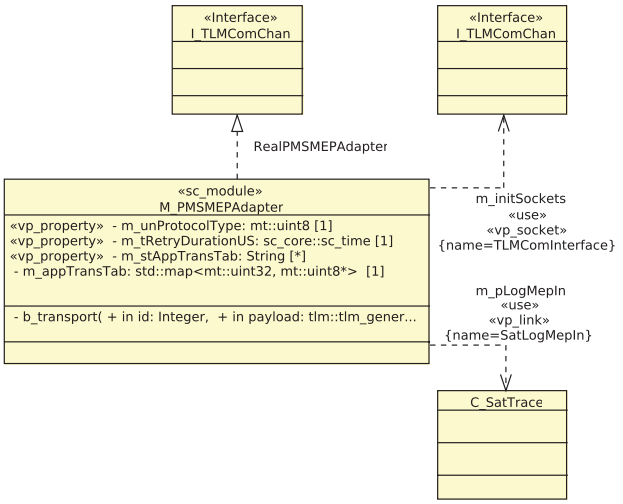

Figure 6: UML class diagram of a basic block

automatic code generation targets two different areas. First, the generation of block skeletons, creating member variables, factory- and linker methods and function stubs. These skeletons are compiled to create the simulation executable. The second task covers the generation of xml configuration files, which are used during runtime to assemble the simulation platform. The basis for the code generation is a set of UML diagrams. The Eclipse Papyrus framework is used as tooling environment. The original framework is extended by Eclipse Plug-ins to accomplish the code generation. Additionally the UML is extended with custom UML profiles.

To generate the *Block Library*, UML class diagrams are used. Each block is presented as single class, with the contained member variables and functions. A custom UML profile allows specifying SystemC characteristics. It is possible to specify whether the generated class is a *sc_module*, *sc_channel* or if a function should be declared as *sc_thread*. A second profile offers the capabilities to specify evaluation framework specific information. It is possible to associate member variables with an ID, to allow the parameterization of this variable with the xml file. Additionally it is possible to extend associations with an ID and a type. The type covers currently pointers, *sc_ports* or TLM sockets. With all these information the block skeleton is generated. The class inheritance, the variable and function declarations and their visibility are taken into account. The factory method implementation, with the parameter handling, is generated. The required information such as the type and the associated ID are specified by the UML diagram. In Figure 6 the basic building block *M_PMSMEPAdapter* is shown. The complete class is stereotyped as *sc_module*, resulting in the creation of a SystemC module. It can be seen that the first three properties are stereotyped as *vp_property*. This stereotype indicates that the created member variables are configured using the xml file. The class is associated with other classes or interfaces. Each association creates a parser entry in the linker method. Dedicated stereotypes allow specifying the type and the ID of the association. With this information the linker method can be generated. Besides, all member variables and function declarations are generated. In the current implementation only the function declaration and an empty function stub is generated. No actual functionality is covered by the code generation. The effort needed to specify the actual functionality graphically equals or exceeds the effort to solve the same task programmatically. Therefore

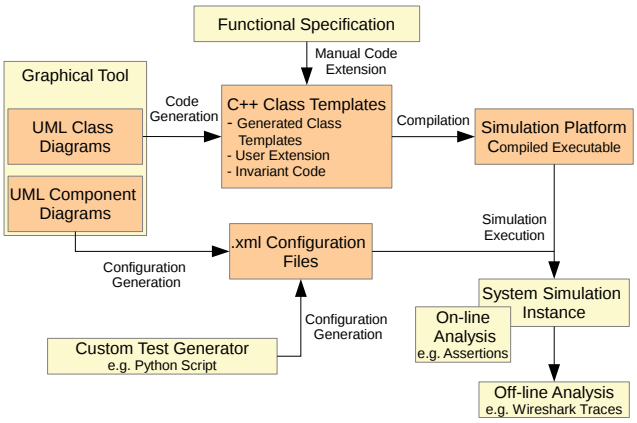

Figure 7: Resulting Analysis Flow

only the function stub is generated and the implementation is done manually. There exists a set of established tools, such as MATLAB/Simulink, supporting the functional code generation. The output of these tools can be used to implement the function stubs. To allow on the one hand an automatic code generation and on the other hand a manual extension of the code, merge techniques known from software versioning are used. For each generated code an ancestor file is maintained that allows detecting differences between a new version of the generated code, the user modifications and the previously generated code. The merging functionality solves non conflicting differences and prompts the user to solve conflicting changes manually. This allows a comfortable side by side editing of the automatically generated and manually extended code.

The second task that is targeted by the graphical tool support is the generation of the xml configuration files. For each block the parameters, associations and the unique IDs are already specified by UML class diagrams. By generating instances of these classes, assigning parameter values to the already specified parameters and interconnecting the instances the xml configuration file can be described. The Papyrus framework offers composite structure diagrams for this task. With the help of these two diagrams block skeletons for the *Block Library*, the *Block Registry* and the xml configuration files are generated automatically.

With code generation support the additional coding overhead caused by the xml configuration approach is canceled out. All necessary code to initialize properties and link blocks is automatically generated, allowing the user to focus on implementing the block functionality. The current extent of the simulation platform covers, e.g., 55 blocks with a total of 212 properties and 87 links.

6.1 Resulting Analysis Flow

The concepts like graphical specification, xml configuration files and a pre-compiled simulation platform results in an analysis flow, shown in Figure 7. In the first phase the class diagrams with the contained parameters and associations are graphically specified. In the next step the block skeletons with the function stubs are generated. Afterwards the user manually implements the desired functionality and compiles the complete simulation platform. Detached from the implementation the system configurations are specified as composite structure diagrams and generated as xml files. The compiled simulation platform and the generated xml files are used to simulate different systems. By changing aspects in the composite structure diagram, e.g., changing parameter values or associations, different systems can be simulated. Besides generating the configuration files graphically, test generators can be used to generate test cases.

There are two general ways of system evaluation. In case of an on-line verification, it is checked during the simulation runtime whether specified properties are met. This can be done using assertions included in the simulation platform. The standard C++ assertions can be used to test basic value dependent specifications. Assertions based on finite linear time temporal logic (FLTL), similar to the approach presented in [12] allow checking one or more complex temporal specifications. During the simulation it is checked whether assertions are violated and in that case the simulation is terminated and an error report is generated. In case of an off-line verification the simulation creates trace files that record the system behavior. After the simulation the correct behavior of the system is verified based on the recorded trace files. This can be done by comparing the system behavior with a set of requirements, a golden reference or with previous simulation traces. The comparison with previous system traces is useful if aspects of a system change and the effects of these changes should be monitored, similar to the reliability analysis presented in [11]. In the current simulation framework different modules exist that generate a variety of trace files. Some traces are suitable for third party tools like Wireshark or the OptoLyzer Suite (Figure 10), enabling the usage of existing tools to analyze the simulated system behavior. Other modules generate custom trace files that are human-readable or that are used with proprietary tools for post-processing, e.g., Python or MATLAB scripts to analyze or visualize traces (Figure 9). Users can easily determine the observation extent by adding the appropriate module in the composite structure diagrams. The tracing extent strongly affects the simulation performance.

7. INTEGRATION OF SW PROTOTYPES

It is necessary that already developed software can be integrated into the virtual prototype, especially in late design phases. This is useful to test the developed software or to refine the behavior of the virtual system to mirror the physical system. In the area of embedded systems the software is often written in ANSI C. Because of the similarity between ANSI C and C++ an integration is nearby, nevertheless some challenges emerge. The main challenge is to encapsulate the ANSI C code, in a way that multiple, none influencing instances are created. Because the virtual prototype describes the complete system, it is most likely that the ANSI C code is executed on different virtual devices. This implies that multiple instances of the ANSI C code are simultaneous present in the simulation environment. Currently three different approaches are implemented.

For small software implementations it is possible to encapsulate the ANSI C functions within C++ class definitions. The complete ANSI C code is copied into a C++ class definition; see Figure 8 (a). Provided interfaces are therefore added to the according class. For the required interfaces a modification of the original code is needed. Global function calls or variable accesses have to be mapped to class member functions or variables. With the help of macros it is possible to integrate the unmodified ANSI C code, by defining substitutions at the beginning of the files. For example if the ANSI C code needs a global function to send data, this call is substituted to call the external class that provides the required member function. The MHP is integrated this way. If the software is more complex this approach isn't feasible. The amount of files and their interdependencies would complicate the approach. Another approach is to encapsulate the

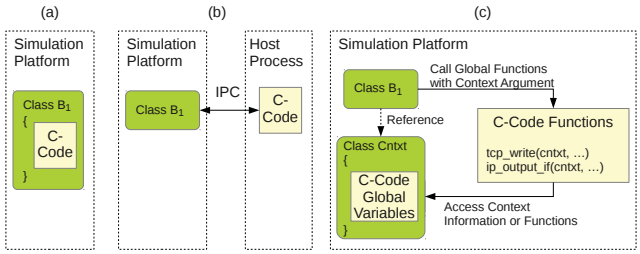

Figure 8: Integration of ANSI C Code

ANSI C code into different host system processes. This way all functions and global variables are encapsulated within one process. Having multiple instances within the virtual system is achieved by creating multiple processes. Based on the fact, that different processes manage their own address space there are no interferences between multiple instances. To interact with the simulation environment the inter-process communication (IPC) of the host system is used, see Figure 8 (b). Function calls, parameters and the return values are serialized and transferred using the IPC. This way a loose coupling between the simulation and the ANSI C code is established.

If a closer integration is needed a direct integration is beneficial. This is the case if simulation dependent instructions are often called or if the software code is annotated with timing information, see [13]. To support multiple instances a context object is created, that contains all global variables from the ANSI C code. The ANSI C functions remain global within the simulation platform and are simply included with the *extern* keyword. Each time a function needs to access global variables, it calls the associated setter or getter function in the context object. Therefore each function has to know the current context object it is currently correlated to. This is achieved by passing a pointer of the context object as argument with each subsequent function call.
The lwIP stack is integrated this way. When a basic building block, normally a C++ class, wants to transmit data through the lwIP stack, it only needs a reference to a *lwIP context* object. This reference is forwarded throughout the stack and is used to access context sensitive information along the data processing chain. The original lwIP functions are modified so that each function has a *lwIP context* argument. After the data processing within the stack is finished, a transmit function in the context object is called to send the data with the standardized TLM-2.0 interfaces. As mentioned earlier the interfaces between layers are realized with TLM-2.0. The simulation platform uses a top down approach, meaning the top module has to poll the underlying modules whether data is present. The lwIP stack on the other hand provides a callback function, which should be called when data is present. To arbitrate between these two strategies the *lwIP context* implements a SystemC process that polls the communication network. If data is received the lwIP callback function is called. Besides managing the context sensitive information, the *lwIP context* implements the required OS functionality, like thread creation, semaphores and message boxes.
This approach provides one drawback. The ANSI C functions are still global within the simulation platform. If parts of the simulation platform use libraries with the same declarations, a linkage problem occurs. The problem occurs, e.g., if the standard TCP/IP library of the host system should be used for inter-process communication. In the current implementation the problem is solved by moving the socket communication of these modules to external libraries which are accessed from the simulation platform.

8. EVALUATIONS

In the following section the developed infrastructure is presented on basis of industrial and artificial use cases. Instead of presenting a single evaluation result in detail, diverse evaluation goals and considerations, e.g., regarding the simulation performance are given. The aim is to highlight the potential of the presented approach and foster the confidence into the developed simulation models.

Performance Evaluation. One area of application is the system performance evaluation. With the help of the virtual prototype, performance characteristics of the system can be evaluated long before physical prototypes are available. This information can be used during the design process to achieve efficient design decisions very early. A requirement of current automotive systems, e.g., is routing data from consumer devices, such as smart phones or laptops, via the in-vehicle network. Most of the available standards in this field use IP based protocols. Therefore IP based protocols find their way into automotive networks. For both fields, automotive networks and IP based communication, design experiences were collected over the last years, but the combination of both creates new design challenges. When both protocols were used in previous projects, the already developed virtual prototypes could be reused during the evaluation of the combined system.
In the following use case, a TCP/IP and MHP connection is established via a MOST network. The designers will face questions like: how will both protocols affect each other, because both share a common communication channel? Which parameterization guarantees a fair bus allocation? Is it even possible to reach a fair bus usage by parameterization or is a different topology necessary? All these questions can be addressed with the help of virtual prototypes. In the following use case the virtual prototype consists of one sender and one receiver device. The sender device establishes a TCP/IP connection via the MOST Ethernet Protocol (MEP) and in parallel a MHP connection. The receiver device provides limited buffer space which is read by the application with a fixed rate. By providing this software timing abstraction, the inherent communication stack timing and the bus timing, different transmissions can be simulated and the resulting performance can be monitored.
With the help of the virtual prototype it is shown that the proposed configuration causes MHP to have a high bus utilization. In the worst case this results in the starvation of the TCP transmission. This is based on the faster retransmissions and the used package size. If both protocols experience a message loss, MHP recovers faster and allocates the shared channel more quickly. This way it is not possible for TCP to share the channel equally. One option is to increase the poll period in the receiver device to prevent message losses, but this only works in dedicated scenarios where the buffers are the bottleneck. Another approach is to change the TCP retransmission and receive window parameters and increase the MHP package gap. These different designs can be simulated with the virtual prototype and the resulting performance can be monitored. Additionally the average transmission performance can be verified with different traffic shapes.
In Figure 9 an excerpt of the trace, created with the original proposed parameterization, is shown. The physical frame utilization (a) is visualized as bars which indicate the payload size contained in one frame at the given point in time. Similarly the outgoing messages of the TCP/IP and MHP protocol are plotted as bars according to their size. Graph (e) shows when messages are dropped because of limited receive buffer space. It can be seen that MHP (c) and TCP/IP

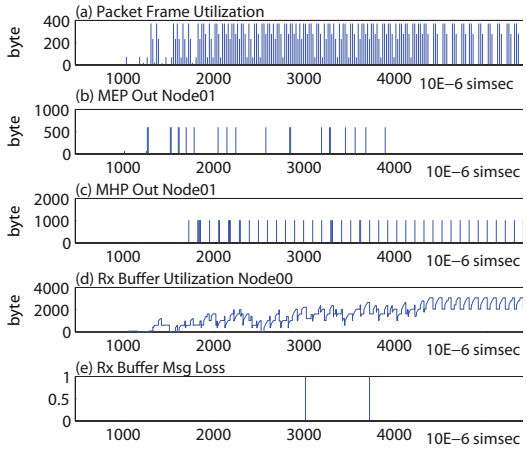

Figure 9: Excerpt of a TCP/IP, MHP trace

over MEP (b) both start to transmit data via a shared channel that offers a suitable bandwidth (a). Both transmissions experience message loss (e), whereas MHP recovers fast and TCP enters a long retransmission time-out, indicated by the pause at the end of (b). The use case illustrated how the virtual prototyping approach offers a detailed insight into the system behavior.

Application Verification. In this section it is shown how an application is analyzed with different degrees of abstraction. A traffic sign recognition (TSR) application is distributed over different computation nodes, using a communication network to exchange data. In a first evaluation the application is mapped to abstract TLM-2.0 channels, meaning the application data is transmitted at once and with an estimated timing for each transmission. In case of the TSR application the data consists mostly of image frames. To verify the behavior of the application, virtual test drives are executed and the algorithm is tested with different input scenarios, e.g., multiple traffic signs in one frame. Different parameters of the algorithm, such as the size and position of the region-of-interest, the radii of the detected circles or the training data of the support vector machine are tested. All scenarios are repeatable, can be paused at any time and all internal states are accessible, improving the functional evaluation of the application and replacing the usual worst case assumptions. Besides the functional verification, aspects such as the duration between capturing a traffic sign and displaying the classified speed value are assessed. Different system parameters, such as application interrupt rates or communication delays are adjusted and the timing effects are analyzed.
During the design process the abstract communication modules, are refined with the target bus technology, such as the MOST bus. After this the same evaluations are executed, but this time with a more accurate communication behavior. Protocol mechanisms of the bus technology, like packet structures, segmentation, retransmission behavior or channel timings are taken into account. Additionally technology specific tools are used to analyze the bus traffic. In Figure 10 a sketch of both evaluation scenarios is given. In the upper part the TSR application uses abstract TLM-2.0 based message queues. The simulation is monitored with custom logging mechanisms, such as video or text output. In a second part the MOST bus and the according protocols are integrated into the simulation. Technology dependent logging tools are used, such as the OptoLyzer tool suite.
In the following the performance of the virtual prototype should be inspected more closely. In Figure 11 the simulation runtime of different application scenarios is shown. The measurements were taken on a simulation host with an Intel®Core™i5-750 processor, 8 GB of RAM and Linux Kernel 3.2.0. The presented applications cover a stereo depth map (SDM) calculation, the already presented TSR application and traffic generators that communicate over MHP respectively TCP. Each scenario is simulated for 60 simsec (simulated seconds) using TLM-2.0 message queues or suitable channels of the MOST bus. The TSR scenario uses the isochronous channel for the camera data, MHP for the cropped signs and the control channel for the classified speed values, see Figure 10. The SDM application uses very complex calculations, in the actual embedded system, dedicated hardware is used for parts of the calculations, therefore the simulation duration is longer than the simulated time. The other use cases are much faster than the simulated time, providing the potential to have an extended test space compared to physical prototypes, e.g., to execute longer virtual test drives. Additionally it can be seen that adding the simulation of the MOST bus increases the simulation duration, because more parallel events have to be simulated. Another parameter that significantly affects the simulation performance is how often the application polls the underlying communication network. The shorter the poll period, the more events are generated. The huge set of parameters makes it difficult to provide a general degree for the performance of the developed simulation platform. Table 1 shows the performance characteristics of the different simulated scenarios. The values represent average values with an average coefficient of variation of about 1.47%, regarding the runtime. When the SDM is executed without the MOST network only a few events occur. This is based on the fact that all calculations regarding a single image are aggregated into one event. Therefore the events are proportional to the exchanged images. The last column shows the relation between the simulated time and the runtime; the smaller the number, the faster the simulation. Only the SDM simulation needs more time than the simulated time, characterized by a value greater than one. The SDM use case has the greatest increase of events, when adding the MOST bus simulation. This is based on the fact that the application transfers the greatest amount of data, which have to be further segmented to be transferred via the MOST bus. Regarding the protocol simulation of TCP/IP and MHP the event increase isn't that significant, because the protocols already segment the data into suitable sizes.

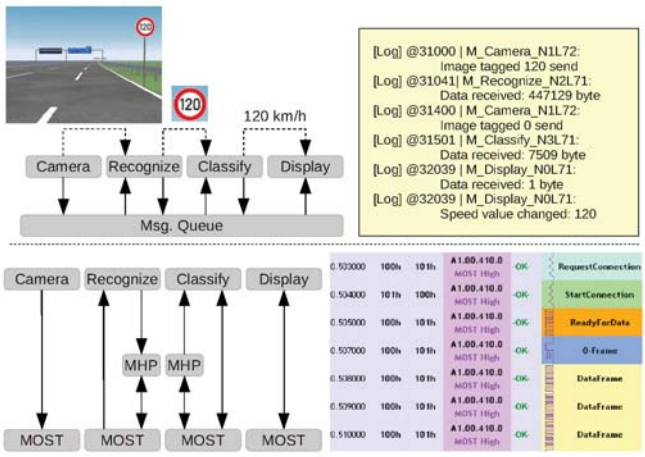

Figure 10: Structure and traces of the TSR scenario

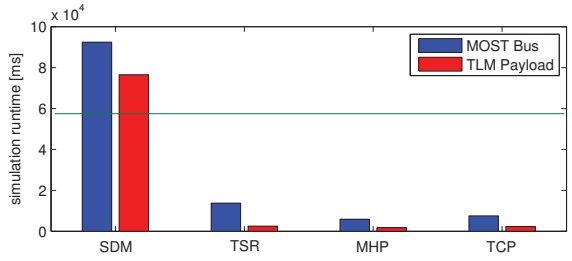

Figure 11: Runtime of different applications

Protocol Verification. The virtual prototype does not only support the application development, but also the communication protocol design. Both the MOST and the FlexRay specification contain different state diagrams specifying the behavior of the communication controller. To verify the correct behavior in an early design stage, these state machines are implemented within the virtual prototype. This allows the evaluation of the protocol design in different bus scenarios. One of the most beneficial evaluations is the design of timers. Both specifications give ranges for different timers, e.g., MOST specifies a maximal duration for the device start-up. With the help of the virtual prototype different bus configurations, amounts of devices, device propagation delays or timer configurations can be simulated. It can be verified that in all scenarios the timer boundaries are not violated. Because the configuration space is still very large, making it infeasible to verify each parameter combination, techniques like corner case analysis are applied.

INET/OMNeT++ Comparison. In the following the developed virtual prototype is compared to the TCP/IP model from the INET/OMNeT++ framework [1]. First the functional equivalence of the two models is compared. Therefore a sender and receiver device is connected with a bandwidth restricted bus. Scenarios with 100 Mbit/s and 60 Mbit/s are simulated. In the upper left part of Figure 12 the transmitted bytes are plotted. The red graphs show the amount of data in the INET model and the blue graphs the bytes of the SystemC based virtual prototype. It can be seen that the graphs are almost identical, increasing the confidence into the model. The small deviation is based on the fact that the two models have a different internal structure. The SystemC based model, e.g., simulates a polling mechanism between the TCP/IP stack and the communication controller, mirroring a transceiver chip with non interrupt-driven I/0 or a fixed service loop. The INET model doesn't have such a polling mechanism, instead the protocol is triggered when data is received. In the upper right graph the distribution of the TCP/IP frames of the 60 Mbit/s scenario is shown. The TCP/IP frames are classified into data frames and sole administrative frames, such as acknowledgements from the

	#ev	ev/simsec	sec/simsec
SDM - MOST	99788424	1663140	1,54141
SDM - TLM	19785	330	1,27548
TSR - MOST	55953138	932552	0,22937
TSR - TLM	36556	609	0,04205
MHP - MOST	25038057	417301	0,09860
MHP - TLM	3711442	61857	0,02991
TCP - MOST	30252391	504207	0,12564
TCP - TLM	8728427	145474	0,03879

Table 1: Performance summary of different models

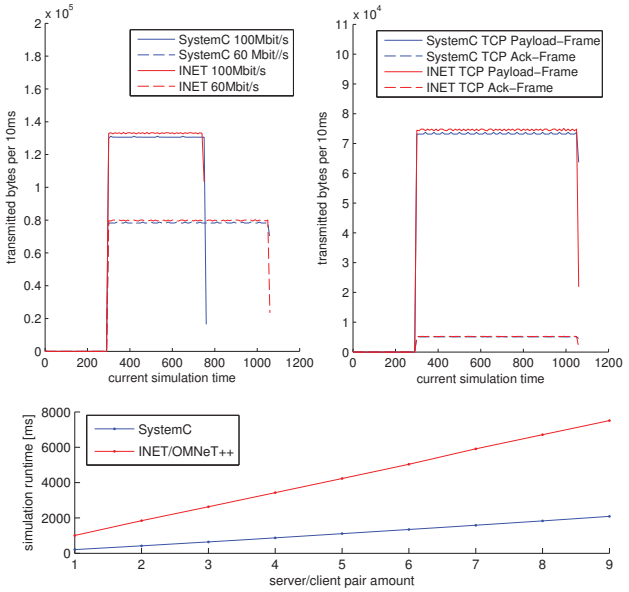

Figure 12: Functional and performance comparison of SystemC and INET/OMNeT++

receiver device. It can be seen that both simulations possess an equal distribution, indicating that both models behave equally. Comparing the two trace files in Wireshark supports this conclusion.

In the second part the simulation runtime of both simulations is compared. The sender/receiver scenario is simulated multiple times in parallel. The monitored simulation time is shown in the lower part of Figure 12. It can be seen that the performance of the proposed virtual prototype is competitive to the INET/OMNeT++ model. In both models the available logging/tracing mechanism are disabled. The OMNeT++ simulation is executed with the command line user interface, minimizing the overhead from the user interface. Both measurements were taken on the same simulation host as the previous section. The simulation time is averaged with 100 repeated simulation runs and shows an average coefficient of variation of about 0.82%.

Besides comparing the performance of different models, in the following the performance of the used simulators is compared. Therefore a synthetic module is implemented that consists of a single loop that periodically executes a set of additions. The amount of additions and the period duration is configurable. Similar to the previous comparison all simulations are executed with the command line user interface and disabled logging. In Figure 13 the results of three different configurations are shown. Each scenario is executed with periods of 0.1, 0.01, 0.001, 0.0001, 0.00001 simsec and a total simulated time of 100 simsec. The first scenario *OMNeT++/SystemC 100* executes a block of 100 additions each period and the second a block of 10000. The scenario *OMNeT++/SystemC 10x100* simulates 10 modules each executing a block of 100 additions in parallel. The monitored runtime is shown in Figure 13. It can be seen that the SystemC simulator is competitive to the OMNeT++ simulator. In all cases SystemC provides the shorter simulation runtime. The presented evaluations show that by choosing SystemC no inherent performance drawbacks are introduced to the simulation framework. It is possible to keep in line with a state of the art general purpose simulation framework such as OMNeT++.

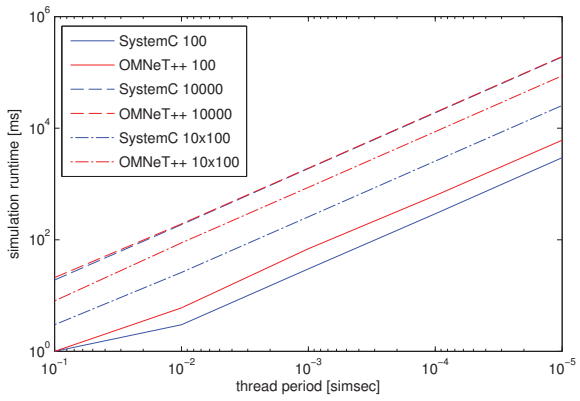

Figure 13: Synthetic comparison of SystemC and OMNeT++

9. CONCLUSION AND OUTLOOK

In this paper a comprehensive approach is demonstrated that uses virtual prototyping to support the analysis and assessment of distributed automotive systems. With the help of software-based system simulations, the design under test is evaluated along the design process. The continuous design support is reached by the refinement capability of the virtual prototype. Based on mechanism like a layered structure, standardized interfaces, modularity and parameterization, a versatility is reached that allows evaluating a variety of system alternatives based on a small set of basic building blocks. This promotes the iterative and incremental development of a comprehensive simulation platform. With the help of graphical tool support the overhead of the proposed infrastructure is canceled out, providing additional support to the user. The presented configuration approach fosters the evaluation automation, decreasing the simulation runtime significantly. Industrial use cases demonstrate how the virtual prototype is used to support what-if analyses, evaluating the influence of different parameters. A TSR use case shows the integration of actual software implementations, allowing the functional and timing verification of software in conjunction with hardware, before physical prototypes are available. Comparing the presented approach with the OMNeT++ framework shows the competitiveness of the approach. The presented concepts are transferable to other simulation tools. The addressing and layering approach, e.g., could encourage a general design of OMNeT++ models. The configuration approach is transferable to other component-based tools.

Extending this framework with generic error stimulation modules and combining the reliability assessment flow presented in [11] with the presented flow will be target of future work. By the combination of these approaches an analysis framework is created that allows to evaluate safety-critical systems in early design phases, taking functional and timing effects onto the complete system into account. It would be possible to change different variables within the simulation, mirroring the possible errors of the physical system, and monitor the effects onto the complete system, taking the error tolerance of the system into account.

Acknowledgement

This work has been partially supported by the German Ministry of Science and Education (BMBF) in the project EffektiV under grant 01IS13022 and the FP7 project OpEneR (Grant No. 285526).

10. REFERENCES

[1] R. Bless and M. Doll. Integration of the freebsd TCP/IP-stack into the discrete event simulator OMNET++. In *Simulation Conference. Proceedings of the 2004 Winter*, 2004.

[2] S. Chai, C. Wu, Y. Li, and Z. Yang. A NoC Simulation and Verification Platform Based on SystemC. In *Computer Science and Software Engineering, 2008 International Conference on*, 2008.

[3] D. B. Chokshi and P. Bhaduri. Performance analysis of FlexRay-based systems using real-time calculus, revisited. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

[4] F. Fummi, D. Quaglia, and F. Stefanni. A SystemC-based framework for modeling and simulation of networked embedded systems. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, 2008.

[5] A. Hagiescu, U. D. Bordoloi, S. Chakraborty, P. Sampath, P. V. V. Ganesan, and S. Ramesh. Performance analysis of FlexRay-based ECU networks. In *Proceedings of the 44th annual Design Automation Conference*, 2007.

[6] IEEE. IEEE Standard for Standard SystemC Language Reference Manual. Sept. 2012.

[7] W. S. Kim, H. A. Kim, J.-H. Ahn, and B. Moon. System-Level Development and Verification of the FlexRay Communication Controller Model Based on SystemC. In *Future Generation Communication and Networking, 2008. FGCN '08. Second International Conference on*, 2008.

[8] A. Lewicki, J. del Prado Pavon, and J. Talayssat. A Virtual Prototype for Bluetooth over Ultra Wide Band System Level Design. In *Design, Automation and Test in Europe, 2008. DATE '08*, 2008.

[9] L. Ouedraogo and R. Kumar. Computation of the Precise Worst-Case Response Time of FlexRay Dynamic Messages. In *Automation Science and Engineering, IEEE Transactions on*, 2013.

[10] R. Pichappan and S. Aziz. A Bus Level SystemC Model for Evaluation of Avionics Mission System Data Bus. In *TENCON 2005 2005 IEEE Region 10*, 2005.

[11] S. Reiter, M. Pressler, A. Viehl, O. Bringmann, and W. Rosenstiel. Reliability assessment of safety-relevant automotive systems in a model-based design flow. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, 2013.

[12] J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued AR-automata. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, 2001.

[13] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, 2011.

[14] A. Sulflow and R. Drechsler. Modeling a Fully Scalable Reed-Solomon Encoder/Decoder over GF(p^m) in SystemC. In *Multiple-Valued Logic, 2007. ISMVL 2007. 37th International Symposium on*, 2007.

[15] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, 2008.

[16] Website. Mentor Graphics®Visual Elite™. http://www.mentor.com.

[17] Website. NetSim hompage. http://tetcos.com/.

[18] Website. NS-3 hompage. http://www.nsnam.org/.

[19] Website. Synopsys DesignWare® IP solutions and Virtualizer™ virtual prototyping tool. http://www.synopsys.com.

[20] Website. Wind River®Simics. http://www.windriver.com.

[21] J. Zimmermann, S. Stattelmann, A. Viehl, O. Bringmann, and W. Rosenstiel. Model-driven virtual prototyping for real-time simulation of distributed embedded systems. In *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, 2012.