

# CONFIGURABLE PROTOCOL ENGINE FOR RUNTIME-CONFIGURABLE COMMUNICATION SUBSYSTEMS ON MULTIPROCESSOR SOC

Petri Kukkala, Marko Hännikäinen and Timo D. Hämäläinen  
Tampere University of Technology, Institute of Digital and Computer Systems  
P.O. Box 553, FI-33101 Tampere, Finland

## ABSTRACT

This paper presents a Configurable Protocol Engine (CPE) to implement runtime-configurable communication subsystems, which are able to adapt their protocol stacks to varying service requirements. The communication subsystems with CPE are designed and implemented using a UML-based design methodology and automated design flow. CPE has been applied to implementing wireless protocol stacks on multiprocessor System-on-Chip (SoC) platforms. As a design case study, we present the implementation of a WSN-to-WLAN bridge on a multiprocessor SoC on FPGA. Experiences with CPE proved its feasibility in rapid implementation of communication subsystems with very decent performance.

## I. INTRODUCTION

This paper presents a Configurable Protocol Engine (CPE), which is targeted at implementing runtime-configurable communication subsystems on multiprocessor System-on-Chip (SoC) platforms. CPE enables runtime-configurability according to *required services* as well as *available platform and network resources* at runtime.

CPE comprises a library of general-purpose atomic protocol functions, which are used to assemble standard as well as customized protocol stacks. Each protocol function encapsulates one typical communication service, such as checksum calculation, data encryption or flow control.

The scope of CPE is illustrated in Fig. 1, which presents two different implementation architectures for multi-mode (multi-radio) communication subsystems. The traditional way is to use layered architecture and implement separate protocol stacks for each type of radio. The redundancy of functionalities on different protocol layers is an issue decreasing the end-system performance and increasing resource usage [1, 2, 3]. With CPE, we can use a single protocol engine to implement the whole communication subsystem.

In this paper we focus on the structure and functionality of CPE. Further, we present the mechanisms that CPE uses to assemble and execute protocols.

The paper is organized as follows. First, Chapter II surveys related research. Chapter III presents the implementation of communication subsystems with CPE. The structure and functionality of CPE is presented in Chapter IV. A design case study is presented in Chapter V and Chapter VI concludes the paper.

## II. RELATED RESEARCH

A Dynamically Assembled Protocol Transformation, Integration and Validation Environment (ADAPTIVE) [4] is an in-

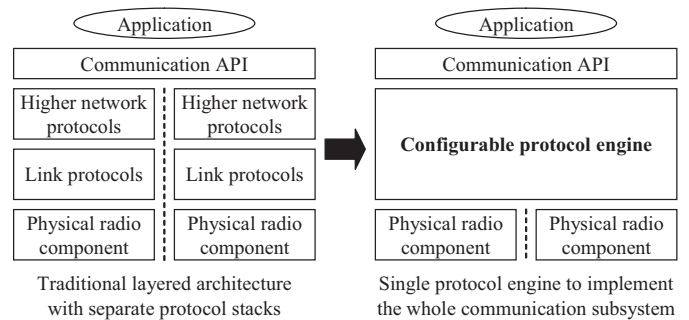


Figure 1: Different implementation architectures for multi-mode communication subsystems.

tegrated framework for protocol composition, evaluation, and experimentation. It decomposes complex protocols into simple protocol functions. Protocol composition is accomplished by combining the functions to form a required transport system. Configuration parameters cover three main areas: requirements for a desired transport system, available hardware resources, and network characteristics. The system is adaptive to changes in the parameters through a reconfiguration of the transport system.

The Dynamic Configuration of Protocols (Da CaPo) [5] is a three-layer model of communication systems for the dynamic configuration of light-weight protocols. The three layers represent the communicating application, end-to-end communication support, and an underlying transport infrastructure. The end-to-end communication support layer is derived according to input parameters including requirements for the communication system, and services available in the transport infrastructure. The layer is composed of protocol modules representing simple communication services [6]. Dependencies between the modules define the required modules to implement a service, and the order in which the modules should be executed.

The Function-based Communication SubSystem (F-CSS) [3] includes a set of protocol functions that are dynamically combined to configure a protocol engine that fulfils the presented requirements for communication services. F-CSS is used to form a whole protocol stack between an application and a network environment. The right combination of protocol functions is selected according to both quantitative and qualitative service requirements. The quantitative requirements cover desired throughput, delay, response time, and jitter. Qualitative requirements specify issues related to session management, stream management, and the manipulation of protocol data units.

Coyote [7] is a framework for implementing modular and

configurable high-level protocols for the customized needs of communication services. With the selection of suitable micro-protocols, and by configuring them together with a runtime system, a composite protocol is constructed. A complete network subsystem can be achieved by combining the composite protocol hierarchically with other protocols [8]. While the Coyote system is primarily designed for configuring micro-protocols at system build time, the adaptation to changes in the environment is done by changing the composition of used micro-protocols.

A role-based architecture [1] uses functional units called roles to organize the communication services. The approach avoids the layering of protocols to achieve more flexible structure and extensibility. Further, the roles are not organized hierarchically, which provides rich interaction between them without the restrictions of protocol layers. In the implementation, roles must have specified ways to be defined and structured. An engine is needed for instantiating and executing the protocols composed by roles.

#### A. CPE Design Objectives

CPE has common features and basic principles with the related protocol configuration systems presented above. The parametrization of protocol functions and resources is in a significant role when automating and optimizing the selection of functions for a required service. By including dependencies between different protocol functions, as in Da CaPo, we are able to reduce the computation in the creation of a protocol configuration.

Available platform resources are taken into account only in ADAPTIVE. While fading the borders between protocol layers – that is the main idea in role-based architectures – an efficient and flexible implementation of several protocols can be created using a common protocol engine.

Contrary to the presented systems, CPE combines the selection of protocol functions with the awareness of available resources, and has a complete engine for the execution of protocols. Consequently, CPE implements communication subsystems especially suitable for embedded wireless network terminals, which have limited resources, but tight requirements for communication services.

The design of CPE and protocol functions utilizes object-oriented approach, and careful partitioning of protocol functionality into reusable and manageable components. The modular structure enables also the flexible development of CPE itself and its components. Further, modularity enables us to efficiently reuse protocol functions, which saves time and effort compared to the development of a complete protocol from the scratch, while dynamic configuration at runtime enables meeting the changing requirements [9].

### III. IMPLEMENTATION OF COMMUNICATION SUBSYSTEMS WITH CPE

Communication subsystems with CPE are implemented using a UML-based Koski design flow [10]. UML is used to design both the general functionality of CPE and the library of protocol functions. UML 2.0 was chosen as a design language based

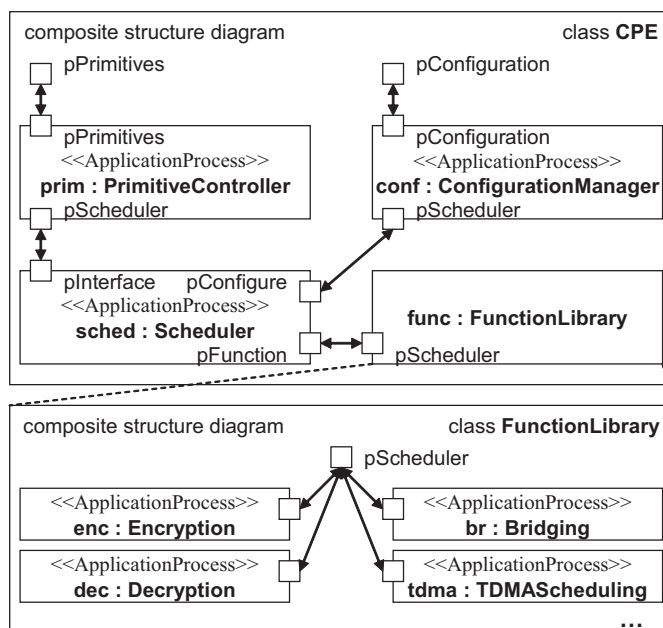


Figure 2: UML 2.0 composite structure diagrams of CPE and function library.

on three main reasons. First, previous experiences have shown that UML suits well the implementation of communication protocols and wireless terminals [11, 12]. Second, UML 2.0 provides formal action semantics and code generation, which enable rapid prototyping. Third, UML is an object-oriented language and supports modular design approach that is an important aspect of CPE.

In Koski, the whole design flow is governed by UML models designed according to a well-defined UML profile for embedded system design, called TUT-Profile [13]. The profile introduces a set of UML stereotypes, which categorize and parameterize model elements to improve design automation both in analysis and implementation. The TUT-Profile divides UML modeling into the design of *application*, *architecture* and *mapping models*.

The application model is independent of an architecture and implements both the functionality and structure of an application. In the TUT-Profile, *application process* is an elementary unit of execution, and they are implemented as asynchronously communicating Extended Finite State Machines (EFSM) using UML statecharts with action semantics. Further, library functions can be called inside the statecharts to enable efficient reuse. When designing a communication subsystem, an application model defines CPE and protocol functions as presented in UML 2.0 composite structure diagrams in Fig. 2. Different components of CPE are considered in details in Chapter IV.

The architecture model is independent of an application, and instantiates the hardware components used by a designed communication subsystem. Hardware components are selected from a platform library that contains available processing elements and communication architectures. Processing elements are general purpose processors as well as dedicated hardware accelerators. The mapping model defines the mapping of CPE

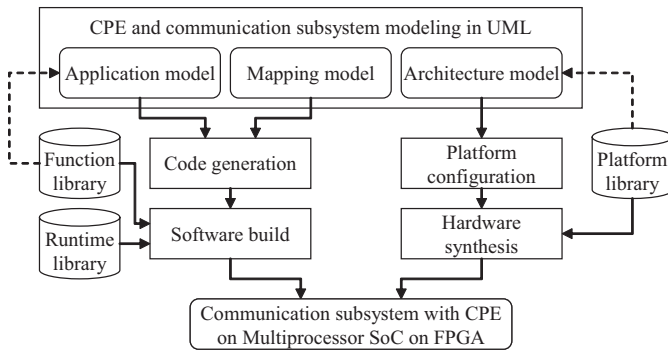


Figure 3: UML-based design flow for the implementation of communication subsystems with CPE.

and protocol functions to the platform, i.e., how application processes are executed on the instantiated processing elements.

Koski enables a fully automated implementation for a multiprocessor SoC on FPGA according to the UML models as presented in Fig. 3. Koski comprises commercial design tools (Telelogic Tau G2, Altera Quartus II) and self-made tools [14]. Based on the application and mapping models, Koski generates code from statecharts, includes library functions and a runtime library, and finally builds distributed software implementing a communication subsystem with CPE. Based on the architecture model, Koski configures the library-based platform and synthesizes the hardware for a multiprocessor SoC on FPGA.

#### IV. STRUCTURE AND FUNCTIONALITY OF CPE

CPE consists of four main components as presented in Fig. 4. The components are a *scheduler*, *primitive controller*, *function library* and *configuration manager*. The communication between CPE and its environment takes place through two interface instances. The *primitive controller* is used for the exchange of *protocol primitives* between CPE and adjacent protocol layers. The second interface is the *configuration interface* that delivers *service requirements* used to configure CPE.

##### A. Function Library

The function library contains a set of protocol functions that are available on a underlying platform. The protocol functions

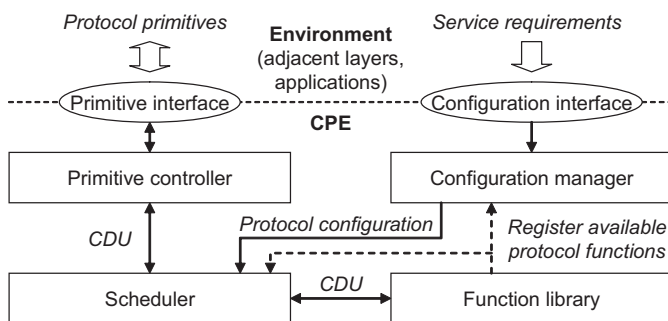


Figure 4: Main components of CPE and its interfaces to an environment.

may implement various kinds of functionality.

Each protocol function implements a class in an application model in UML. At start-up, each protocol function registers for the scheduler and configuration manager. Consequently, the scheduler and configuration manager are aware of available protocol functions and how to access them.

##### B. Configuration Mechanisms of CPE

The runtime-configuration of CPE contains two phases. First, service requirements are delivered to the configuration manager, which analyzes the requirements and fit them to the available platform and network resources. Second, the configuration manager creates a *protocol configuration* meeting the placed requirements, and sends the protocol configuration to the scheduler.

The service requirements define protocol stacks that CPE is expected to implement. There are two complementary approaches how we may define the desired stacks. First, we can define certain *types of protocol functions* that must be included to a configuration, such as we need encryption, error correction and flow control. Second, we may explicitly define *exact protocol functions*. Thus, we may define that we want to use Advanced Encryption System (AES) algorithm for encryption and Cyclic Redundancy Check (CRC) algorithm for error detection. In the first case, the configuration manager may select predefined protocol stacks according to the rules that are defined at design-time.

A protocol configuration specifies a *processing sequences* for different types of *CPE Data Units* (CDU). CDU is an internal data structure that is used when processing protocol primitives in CPE. Each CDU is of a certain type according to the types of protocol primitives. A processing sequence defines which protocol functions are called, and in which order, for a certain type of CDU. Further, a protocol function may also terminate as well as initiate a processing sequence.

The methods and algorithms to optimize the configuration at runtime belong to the future work. We are developing methods that are aware of Quality-of-Service (QoS) and realtime requirements.

##### C. Processing the Protocol Primitives

The primitive controller constructs CDUs from the primitives received through the primitive interface. Further, the controller constructs primitives from CDUs received from the scheduler. The CDUs and primitives are sent to the scheduler and primitive interface, respectively.

The scheduler controls the processing of CDUs on the protocol functions in the function library. The scheduling is performed according to processing sequences defined a protocol configuration. When the scheduler receives CDU, it checks the type and phase of CDU, and resolves next protocol function that should process CDU. CDU is delivered to the function, which processes CDU and returns it back to the scheduler. Each CDU is repeatedly scheduled to the protocol functions until a processing sequence is finished, in which case, CDU is sent to the primitive controller.

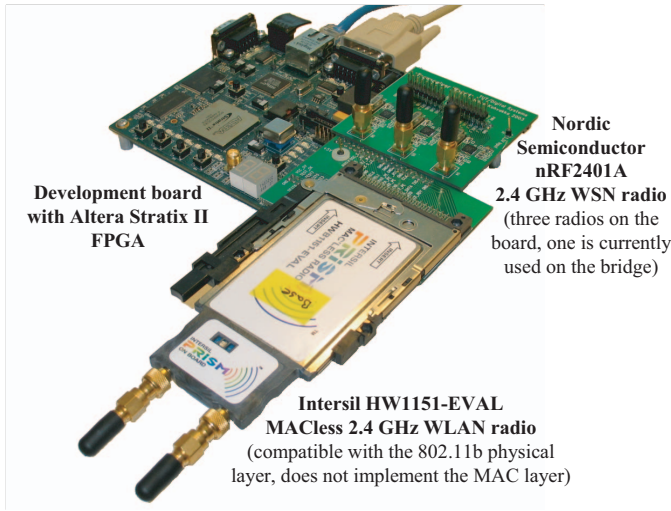


Figure 5: Development board with extension cards for WLAN and WSN radios.

The scheduler is designed in such a way that in a distributed multiprocessor implementation each processor may execute a local copy of the scheduler, which schedules the processes active in a processor. The local scheduling is operated independently, and no master scheduler is needed. This approach reduces the Inter-Processor Communication (IPC) significantly, since IPC is required only when consecutive protocol functions in a processing sequence are mapped to different processors. Further, the implementation and execution overheads caused by the local copies of the scheduler can be considered negligible.

### V. DESIGN CASE STUDY – WSN-TO-WLAN BRIDGE

We evaluated CPE by implementing a terminal that bridge packets between WSN and WLAN on FPGA. The physical hardware platform is a development board with Altera Stratix II (EP2S60) FPGA and extension cards for Intersil MACless WLAN radio and Nordic WSN radio. A photo of the board with radio cards is presented in Fig. 5.

The WLAN radio is 2.4 GHz Intersil HW1151-EVAL MACless radio transceiver, which implements the physical layer of 802.11b, but not the Medium Access Control (MAC) layer. The WLAN radio can be used to with standard 802.11b WLANs as well as customized WLANs, such as TUTWLAN [15]. The WSN radio is 2.4 GHz Nordic Semiconductor nRF2401A narrow band radio transceiver, which comprise the physical layers compatible with ZigBee, Bluetooth and various WSNs.

#### A. WSN-to-WLAN Bridge Implementation

The WSN-to-WLAN bridge is a multi-mode terminal with two different radio interfaces. The bridge has a protocol stack that is compatible with customized WSN and WLAN protocols. The stack contains (i) bridging of data packets between WSN and WLAN, (ii) AES encryption, integrity check and fragmentation of data packets, (iii) assembly and error detection (CRC-8, CRC-32) of MAC frames, and (iv) Time Division Multi-

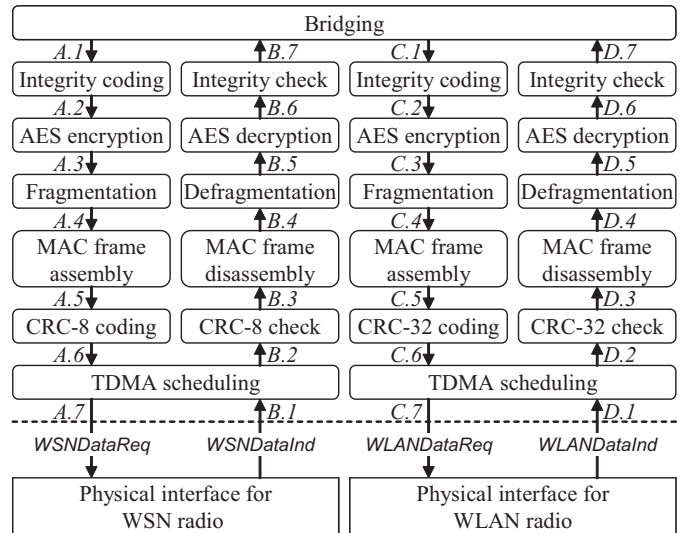


Figure 6: Protocol functions and processing sequences (A.x-D.x) in the WSN-to-WLAN bridge implemented using CPE.

ple Access (TDMA) scheduling that controls the access to the shared wireless media of WSN and WLAN.

CPE was used to create a communication subsystem that implements the protocol stack for the bridge. The implementation of the bridge contained two main steps. First, we defined desired functionality for the bridge, and ensured that the function library contains appropriate protocol functions. In the case we would lack certain protocol functions, we would have to supplement the library or find substitutive functions. Second, we designed corresponding service requirements that are used to assemble the required protocol stack at runtime.

Fig. 6 presents the protocol functions and processing sequences in the bridge. There are four processing sequences (A.x-D.x) corresponding the four protocol primitives that are provided to the physical interfaces for WSN radio (*WSNDataReq*, *WSNDataInd*) and WLAN radio (*WLANDataReq*, *WLANDataInd*). In this case study, the exact protocol functions have been specified in the service requirements. The function library contains the required protocol functions.

#### B. Hardware Platform

Our multiprocessor SoC platform contains up to four Nios II processors for protocol execution and dedicated hardware modules, such as hardware accelerators and interfaces to external devices [16]. These coarse-grain Intellectual Property (IP) blocks are connected using the Heterogeneous IP Block Interconnection (HIBI) on-chip communication architecture [17]. Each processor module is self-contained and contains Nios II processor core, timer units, cache and memory.

The multiprocessor platform on FPGA is presented in Fig. 7. The platform implements hardware accelerators for AES and CRC-32 algorithms, and the WLAN and WSN radio interfaces implement a full hardware interface to access the radios on the development board. Further, the figure presents the mapping of protocol functions to the processors and hardware accelerators.

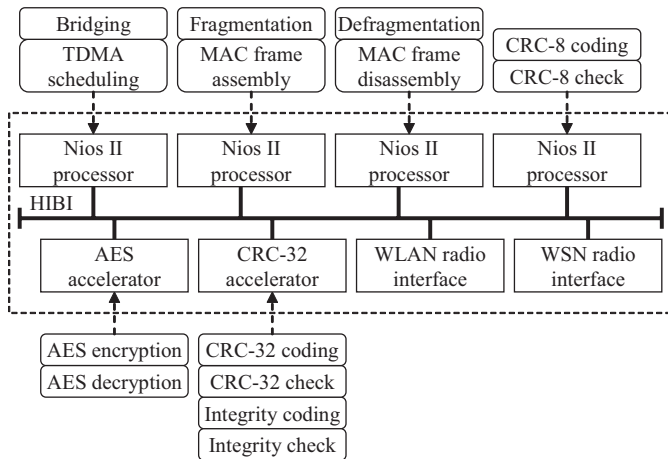


Figure 7: Mapping of the protocol functions to the multiprocessor SoC platform with hardware accelerators.

Currently, the mapping is done manually, but the configuration manager may perform the mapping automatically according to the service requirements and available resources.

### C. Results and Experiences

The software implementation of the bridge was distributed over four processors. The memory usage is 230 kB (113 kB for code and 117 kB for data) per processor. The size of the hardware platform is 29,259 Adaptive Look-up Tables (ALUTs), which take 60% of the total capacity of the used FPGA.

The bridging delay of the presented WSN-to-WLAN bridge was 20–25 ms. This is very decent result when comparing it with our previous protocol implementations on the same hardware platform [14]. Consequently, the overhead caused by CPE is very reasonable.

The experiences with CPE proved that it provides a feasible and efficient approach to implement communication subsystems. The support for multiprocessor implementations enables scalability and fulfils performance requirements.

## VI. CONCLUSIONS

CPE presents a novel UML-based approach to design and implement communication subsystems for embedded wireless terminals. The subsystems with CPE are implemented using the UML-based design methodology and fully automated Koski design flow. The key features of CPE are runtime-configurability and modular structure that enables a high-degree of reuse in design and a flexible usage of SoCs for wireless communications.

This paper focused on the structure and functionality of CPE. We presented how CPE assembles and executes protocols. The future work includes developing configuration and optimization approaches to improve the configuration manager and its awareness of platform resources. Further, we will implement different communication subsystems with CPE to provide wide range of wireless communication services for embedded applications.

## REFERENCES

- [1] R. Braden, T. Faber, and M. Handley, "From protocol stack to protocol heap - role-based architecture," *SIGCOMM Computer Communications Review*, vol. 33, no. 1, pp. 17–22, Jan. 2003.
- [2] V. Srivastava and M. Motani, "Cross-layer design: A survey and the road ahead," *IEEE Communications Magazine*, vol. 43, no. 12, pp. 112–119, Dec. 2005.
- [3] M. Zitterbart, B. Stiller, and A. Tantawy, "A model for flexible high-performance communication subsystems," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 507–518, May 1993.
- [4] D. Box, D. Schmidt, and T. Suda, "ADAPTIVE: An object-oriented framework for flexible and adaptive communication protocols," in *Proceedings of the 4th IFIP Conference on High Performance Networking*, 1992, pp. 367–382.
- [5] T. Plagemann, B. Plattner, M. Vogt, and T. Walter, "A model for dynamic configuration of light-weight protocols," in *Proceedings of the 3rd Workshop on Future Trends of Distributed Computing Systems*, 1992, pp. 100–106.
- [6] B. Stiller and T. Plagemann, "Protocol configuration and interoperability - a case study," in *Proceedings of the IEEE Singapore International Conference on Networks*, 1995, pp. 299–303.
- [7] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu, "Coyote: a system for constructing fine-grain configurable communication services," *ACM Transactions on Computer Systems*, vol. 16, no. 4, pp. 321–366, Nov. 1998.
- [8] N. Hutchinson and L. Peterson, "The x-kernel: an architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, Jan. 1991.
- [9] S. Böcking, "Object-oriented network protocols," in *Proceedings of the IEEE INFOCOM - 16th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1997, pp. 1245–1252.
- [10] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "UML-based multiprocessor SoC design framework," *ACM Transactions on Embedded Computing Systems*, 2006, accepted.
- [11] P. Kukkala, M. Hännikäinen, and T. Hämäläinen, "UML 2.0 implementation of an embedded WLAN protocol," in *Proceedings of the 15th International Symposium on Personal, Indoor and Mobile Radio Communications*, vol. 2, Sept. 2004, pp. 1158–1162.
- [12] —, "Design and implementation of a WLAN terminal using a UML 2.0 based design flow," in *Lecture Notes in Computer Science*, vol. 3553, July 2005, pp. 404–413.
- [13] P. Kukkala, J. Riihimäki, M. Hännikäinen, T. Hämäläinen, and K. Kronlöf, "UML 2.0 profile for embedded system design," in *Proceedings of the Design, Automation and Test in Europe*, vol. 2, Mar. 2005, pp. 710–715.
- [14] M. Setälä, P. Kukkala, T. Arpinen, M. Hännikäinen, and T. Hämäläinen, "Automated distribution of UML 2.0 designed applications to a configurable multiprocessor platform," in *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation*, July 2006, accepted.
- [15] M. Hännikäinen, T. Lavikko, P. Kukkala, and T. Hämäläinen, "TUTWLAN - QoS supporting wireless network," *Telecommunication Systems - Modelling, Analysis, Design and Management*, vol. 23, no. 3,4, pp. 297–333, 2003.
- [16] T. Arpinen, P. Kukkala, E. Salminen, M. Hännikäinen, and T. Hämäläinen, "Multiprocessor platform with RTOS for distributed execution of UML 2.0 designed applications," in *Proceedings of the Design, Automation and Test in Europe*, Mar. 2006, pp. 1324–1329.
- [17] E. Salminen, V. Lahtinen, T. Kangas, J. Riihimäki, K. Kuusilinna, and T. Hämäläinen, "HIBI v.2 communication network for system-on-chip," in *Proceedings of the International Workshop on Systems, Architectures, Modeling and Simulation*, July 2004, pp. 413–422.