Automated Generation of Explicit Connectors for Component Based Hardware/Software Interaction in Embedded Real-Time Systems

Wolfgang Forster[1] Christof Kutschera[2] Andreas Steininger[1] Karl M. Göschka[1]

[1]Vienna University of Technology
Karlsplatz 13, A-1040 Vienna, Austria
{wolfgang.forster,andreas.steininger,karl.goeschka}@tuwien.ac.at

[2] University of Applied Sciences Technikum Vienna
Department of Embedded Systems
Höchstädtplatz 5, A-1200 Vienna, Austria
kutschera@technikum-wien.at

Abstract

The complexity of today's embedded real-time systems is continuously growing with high demands on dependability, resource-efficiency, and reusability. Two solution approaches address these needs: First, in the component based software engineering (CBSE) paradigm, software is decomposed into self-contained components with explicit interactions and context dependencies. Connectors represent the abstraction of interactions between these components. Second, components can be shifted from software to reconfigurable hardware, typically field programmable gate arrays (FPGAs), in order to meet real-time constraints. This paper proposes a component-based concept to support efficient hardware/software co-design: A hardware component together with the hardware/software connector can seamlessly replace a software component with the same functionality, while the particularities of the alternative interaction are encapsulated in the component connector. Our approach provides for tools that can generate all necessary interaction mechanisms between hardware and software components. A proof-of-concept application demonstrates the advantages of our concept: Rapid change and comparison of different partitioning decisions due to automated and faultless generation of the hardware/software connectors.

Keywords: HW/SW interaction, CBSE, embedded real-time systems, automated design flow

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

1. Introduction and Related Work

The importance of embedded real-time systems has rapidly grown over the last years. This trend is also clearly visible in our research area, namely automotive embedded systems domain, especially hard real-time systems for by-wire applications. State-of-the-art vehicles contain up to 70 electronic control units (ECUs) [13] and most innovations in cars are realized in software. Future applications will be based on standardized architectures like AUTOSAR (AUTomotive Open System ARchitecture) [1, 14], which was developed by automobile manufacturers and suppliers. Important drivers for this development are safety, software reuse and cost reduction due to fewer development cycles. AUTOSAR proposes a layered architecture of basic software modules comprising communication modules, operating system and modules facilitating the access to microcontroller peripheral devices as well as a component based infrastructure for application components.

Our concept to support rapid and faultless hardware/software partitioning is based on the component based software engineering (CBSE) paradigm, nevertheless with some few adaptations it can be also applied to existing layered architectures. CBSE is already a widely accepted and adopted paradigm in the embedded systems domain to cope with the increasing complexity. In accordance with the work of [15, 21, 23] a component is defined as (i) a trusted software element, (ii) an element of execution, with a (iii) well defined usage description. It conforms to a (iv) component model and can be (v) independently deployed and composed without modification according to a composition standard.

The aim of hardware/software partitioning methods in real-time embedded systems is to achieve an overall solution that meets all application requirements, which is often quite challenging for tasks that are very time or resource critical. The foundation for partitioning is a detailed high-level description of the complete behavior of an application. Furthermore accurate estimations of important characteristics are of major importance for the process of optimization and partitioning. Promising hardware/software partitioning approaches have been already presented in [3, 4, 7]. The foundation for the co-design algorithms proposed there is an independent high-level specification of the complete behavior. Established high-level specification languages for this purpose are SystemC or ESTEREL. A formal verification of hardware/software partitioning with SystemC can be found in [18]. The partitioning process is constraint driven concerning performance, dependability and cost properties. Nevertheless the final partitioning decision is based on expert knowledge and has proven to be very difficult to automate [24]. A further hardware/software partitioning algorithm concerning hardware area and processor memory constraints is presented in [12]. The approach is classified into a partitioning phase and a scheduling phase. During the partitioning phase the application program is transferred into directed acyclic graphs (DAGs). In the scheduling phase the reduction of system cost will be reached by tuning the tasks in hardware and software. Contrary to our approach the partitioning optimization is based on tasks and not on the level of well specified components and their interactions.

A novel hardware/software co-design approach from Altera[1] is presented in [19]. It describes the process of *Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions*. It is a methodology that is tightly coupled with an Altera *soft core CPU* (used in an FPGA), its tool chain, and its memory system. A C-to-hardware compiler is used to generate hardware accelerators with direct CPU memory access from standard C functions. A major exclusion from this automated process are recursions and floating-point types. The *HybridThreads Project* [17] is a further co-design approach. It is an embedded real-time operating system, that allows programmers to run threads simultaneously on the CPU and in parallel on an FPGA. The thread scheduling mechanism is also realized in hardware to reduce thread switching overheads and jitter. In both approaches the partitioning decision itself is still based on human expert knowledge.

Our approach extends the principle of CBSE towards hardware components such that hardware/software partitioning can be performed with a high degree of automation. This not only reduces the risk of error during the partitioning process (by reducing the amount of required human intervention) but also allows a more systematic treatment. The partitioning method is platform independent and requires only a CPU with access to an FPGA. It is possible to generate interfaces between hardware and software components automatically as well as interfaces between two hardware components. Furthermore, components can be also deployed to hardware in the case that a hardware component requires functionality from a software component — the initiator of a method call is deployed to hardware. Based on our proof-of-concept application we will discuss and demonstrate the challenges of supporting efficient hardware/software partitioning.

[1] http://www.altera.com/

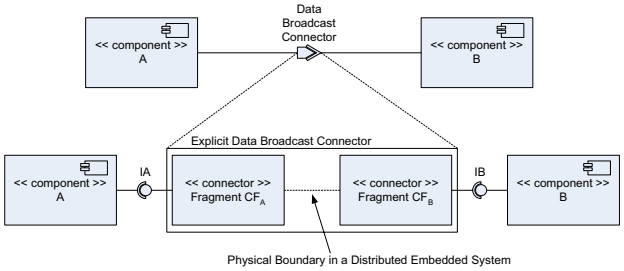

Figure 1. Explicit Data Broadcast Connector [21]

2. Component Model

A component has to provide at least one interface to specify the dependencies between the services provided by the component and the services required to fulfill its task. An interface is a named set of services provided or required by a component. Two components can interact during runtime if their *provided* and *required* interfaces are validly associated. The association is called a connector, which is the abstraction of any interaction between the connected components. We distinguish between two different types of connectors, namely implicit and explicit. Most component models cover the process of component interaction in some kind of middleware, thus the connection has to be considered as implicit connector. An explicit connector is an architectural entity that is used to represent component composition and interaction, whereas the explicit connector itself can be identified as a communication component. Furthermore, an explicit connector encapsulates all communication logic for one specific type of interaction. The use of explicit connectors is especially important if the CBSE paradigm is applied to distributed embedded real-time systems where interaction between components has to be processed via communication systems like FlexRay [8, 11]. In [21] the UML 2.0 Superstructure specification [20] is used and extended for the representation of component di-

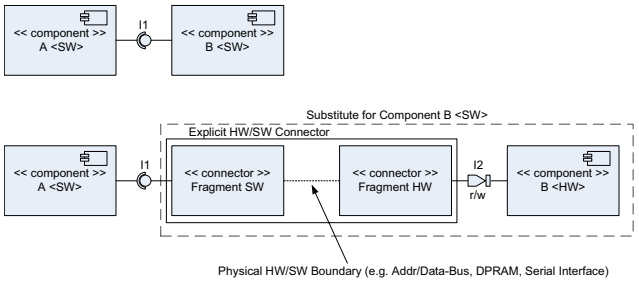

Figure 2. Explicit Hardware/Software Interaction Concept

agrams that contain explicit connectors. As already mentioned above an explicit connector is the representation of component interactions. However, this fact is only valid for platform independent models. In platform specific models an explicit connector has to be transformed into two distributed fragments of a connector. Both fragments represent the total functionality of the original connector as shown in Figure 1. The process of explicit connector transformation is described in [22]. Independent of the use of the connectors — that can be local or distributed between processing units — component model transformation results in local communication interactions of components to the fragments of the explicit connectors, the so-called communication middleware. Furthermore all local communication interactions are represented by synchronous (blocking) or asynchronous (non-blocking) method calls [21] (illustrated by the transformation of the *Data Broadcast Connector* to *IA*,*IB* and the explicit connector in Figure 1).

2.1. Hardware/Software Interaction Model

For our concept of automated generation of hardware/software connectors we extend the use of explicit connectors. In [21] the physical boundary between the connector fragments is a communication system between different electronic control units (ECU's). In our approach the physical boundary is the physical interface between a central processing unit (CPU) or a microcontroller unit (MCU) to a reconfigurable hardware device. Of course, applying hardware/software co-design principles implies that corresponding ECU's are equipped with such a reconfigurable device. *Field programmable gate arrays* (FPGA's) are already state-of-the-art reconfigurable devices in consumer electronics and FPGA's in conjunction with modern MCU cores are viewed as a promising solution for the problem of continuously increasing performance requirements of modern embedded real-time applications.

Figure 2 shows the basic concept of our component based hardware/software partitioning approach. The realization of component *B* in software can be replaced by a hardware implementation of *B*, if the connector is adopted accordingly: An explicit connector is introduced of which one connector fragment has to be realized in hardware and the other one in software. Both together encapsulate the physical boundary between an MCU and an FPGA. Examples for this physical boundary are standard types like address/data bus, serial interface, dual ported RAM, or single wire interface. In Figure 2 interface *I1* is independent of the realization of component *B*, which is an important property of our approach. As already stated above, interface *I1* can only consist of synchronous or asynchronous method calls. For simplification the shown component diagram is only modeled with a synchronous method call. An asynchronous call would also contain a path for a callback as shown in Figure 4. Interface *I2* represents the access from the hardware connector fragment to the effective hardware implementation of the component's functionality. The simplest solution for such a connector is a memory access — also called blackboard connector — which provides the possibility of read and write accesses to the hardware component (indicated by the *r/w* connector in Figure 2). Starting from a given interface specification our approach enables the automated generation of all necessary mechanisms for the replacement of a software component by an equivalent hardware implementation. Of course it is also possible to shift more than one component into hardware. To hide the actual implementation of the interface all specific properties are encapsulated in the connector fragments of the synchronous and asynchronous method calls — as specified in the following sections.

2.2. Synchronous Software Connector Fragment

During a synchronous method call the thread of execution changes from the component with the *required interface* to the component with the *provided interface* until the latter has finished its task. As shown in Figure 3 the software fragment of an explicit synchronous hardware/software connector consists of two sub-components:

Parameter Handler: The *Parameter Handler Component* converts the parameters of the synchronous method calls into data elements that can be transferred via the physical boundary to the hardware and thus triggers the operation of the hardware module of the component. The thread of execution remains at the *Parameter Handler Component* to check continuously the status of the hardware until it has finished the execution. A read back of the results and a parameter conversion complete the synchronous method call and the thread of execution returns to the initially calling component.

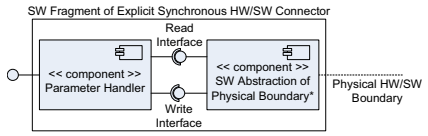

Figure 3. Synchronous Software Connector Fragment

Software Abstraction of Physical Boundary: The *Software Abstraction of the Physical Boundary Component* provides a well defined and simple interface to the hardware devices of an ECU via read and write accesses.

All components marked with a (*) are of type singleton and are deployed only once on an ECU even in case that more than one component has been shifted to hardware.

2.3. Asynchronous Software Connector Fragment

An asynchronous method call implies the possibility of more than one thread of execution on a CPU or MCU. During the asynchronous call the thread of execution changes from the initial calling component to the *Request Handler Component* (see Figure 4) and returns immediately. A second thread of execution is processed cyclically or triggered by an interrupt at the software fragment of the connector. As shown in Figure 4, the software fragment of an explicit asynchronous hardware/software connector consists of four sub-components. Additionally an interrupt or timer service is needed for the realization of this component, which is indicated as *Timer and Interrupt Services Component*. The sub-components of the asynchronous software fragment are:

Request Handler: The *Request Handler Component* has to manage the possibility of multiple asynchronous method calls. It has to forward method calls to the *Parameter Handler Component* via the *Method Call Interface* and will itself be notified about completion of hardware tasks via the *Call Notification Interface*. In the latter case the *Request Handler Component* has to inform the initially calling component via the *Callback Interface*.

Parameter Handler: Same as described in Section 2.2. In addition it has to register each method call at the *Hardware Status Check Component*. In case of a completed hardware task the *Hardware Status Check Component* transfers the results from the hardware to the *Parameter Handler Component* via the *Call Status Notification Interface*. Finally a parameter conversion of the hardware results and a notification of the *Request Handler Component* are necessary.

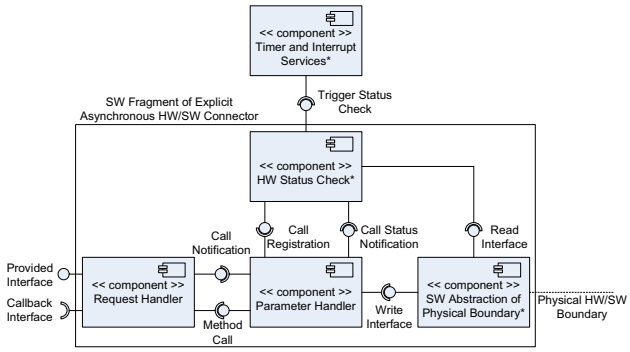

Figure 4. Asynchronous Software Connector Fragment

Software Abstraction of Physical Boundary: Same as described in Section 2.2.

Hardware Status Check: The *Hardware Status Check Component* has to observe the status of all active hardware tasks that have been registered via the *Call Registration Interface*. The activation of the hardware status check has to be initiated by a periodically scheduled task of the operating system or by an interrupt service. The latter can even be triggered by the hardware part of the component itself. In case of a completed hardware task the *Hardware Status Check Component* has to read back the results via the *Software Abstraction of Physical Boundary Component* and transfer them to the *Parameter Handler Component*.

2.4. Hardware Connector Fragment

The hardware connector fragment of an explicit hardware/software partitioning connector is equal for both synchronous and asynchronous method calls. As shown in Figure 5 it consists of two sub-components, whereas the interaction in-between is always of type “blackboard” with the possibility of read and write accesses.

Hardware Abstraction of Physical Boundary: The *Hardware Abstraction of Physical Boundary Component* is the counterpart to the *Software Abstraction of Physical Boundary Component* and provides a well defined and simple interface for read and write accesses.

Hardware Component Handler: The *Hardware Component Handler* has to manage the possibility of multiple

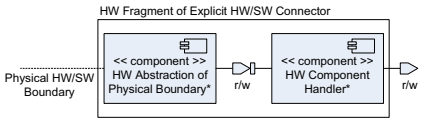

Figure 5. Hardware Connector Fragment

explicit hardware/software connectors and thus multiple realizations of hardware components. All read and write accesses have to be redirected to the corresponding hardware realization of a component. Both the *Hardware Component Handler* and the *Hardware Abstraction of Physical Boundary Component* are of type singleton and are deployed only once in the reconfigurable hardware during the component model transformation process.

All considerations up to now have been made under the assumption that the initial calling component is implemented in software. If the initial calling component of an interaction is realized in hardware (and the called component in software), the software fragments of the explicit connectors have to be slightly modified. In the case of a synchronous method call the *Parameter Handler Component* of Figure 3 has to be activated by a periodically scheduled task of the operating system or by an interrupt service to check the status of the hardware component and to initiate the call. The interface of the software fragment is changed from a provided to a required interface. In the case of an asynchronous method call the assembly of the software connector fragment in Figure 4 can remain unmodified, except the direction of the provided and the callback interface have to be changed.

3. Hardware/Software Interaction

The grand challenge of hardware/software co-design is the design partitioning decision. A widely known approach for this purpose is POLIS [4, 24] from the University of California, Berkeley. The high-level system specification is based on existing languages, e.g. ESTEREL [5] or SystemC [16]. Design partitioning is a process on system level and includes the hardware/software partitioning, the target architecture selection and the scheduler configuration. These decisions are based on expert knowledge in system design and are very difficult to automate [24]. Consequently POLIS can be seen as a supporting tool, the complex decision of hardware/software partitioning itself is still a task for a human engineer and can be supported by consideration of measures and metrics for real-time system components as proposed in [10].

As already mentioned existing hardware/software partitioning approaches are based on the high-level specification of a certain functionality to produce modules (or components) that can be either executed in a software or hardware environment. Our concept extends this approach and closes the interfacing gap if hardware components are deployed: We focus on the automated generation of all necessary interaction mechanisms between hardware and software. According to our *Hardware/Software Interaction Model* only synchronous (blocking) or asynchronous (non-blocking) method calls have to be considered for interaction between a hardware component and the corresponding connector fragment. Thus a simple and generic hardware interface has to be introduced to support the process of automated connector generation. We propose a hardware interface — marked as *I2* in Figure 2 —that contains a (i) control register, a (ii) memory for input parameters and a (iii) memory for output parameters for each component method. This hardware interface can be implemented as a simple memory based interface, containing: (i) *data bus*, (ii) *address bus*, (iii) *access enable signal* and (iv) *read/write control signal*. To support the automated generation of hardware/software connector fragments we have defined the following interface design attributes:

1. Definition of the hardware component architecture (e.g. 16 bit data bus and 6 bit address bus).
2. Control and status information bits for each method of the component interface in the control register of the hardware implementation:
   - start method call
   - abort method call
   - method call active
   - method call ready
   - parameter mismatch
3. Definition of input and output parameter width and memory allocation, which is especially important if a parameter of a method is defined as a pointer in the interface specification.

The resulting attributes of a hardware component interface — as listed above — and the software interface specification are the foundation for the automated connector fragment generation. The software connector fragment will have the same interface as the general software implementation of a dedicated component. From other components' point of view there is no difference between the interface of a software or a hardware realization of a component.

4. Implementation Aspects

To verify our concept of automated generation of the necessary hardware/software interaction mechanisms we

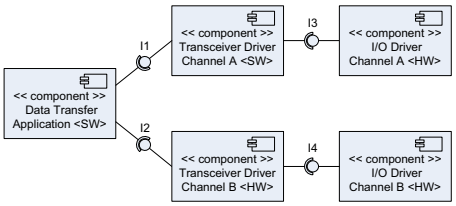

Figure 6. Deployed Component Model for ECU 1

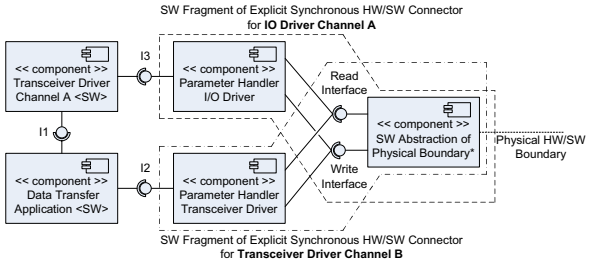

Figure 7. SW Connector Fragments ECU 1

have implemented a prototype proof-of-concept application. The functionality of this application is very simple: Copying data packets from one ECU to another via a FlexRay communication system. Additionally the transceiver driver devices of both FlexRay communication channels have to be controlled. Figure 6 shows the application that was deployed to ECU 1. Each transceiver driver component controls an external FlexRay transceiver device via an input/output (I/O) driver component and checks its status information. The components *Data Transfer Application* and *Transceiver Driver Channel A* have been deployed as software components, whereas the components *Transceiver Driver Channel B*, *I/O Driver Channel A* and *I/O Driver Channel B* have been realized in hardware.

This application has been chosen because of the low-speed interface of the transceiver driver devices, which result in additional scheduling overheads to fulfill the precise timing requirements. In our prototype implementation we compare the resource usage of a software and a hardware implementation of the *AUTOSAR FlexRay Transceiver Driver Module* [2]. Figure 7 illustrates the deployment of the software components and the software connector fragments of ECU 1 after the component model transformation in consideration of the proposed hardware/software partitioning approach. The component specifications of *Transceiver Driver Channel B* and *I/O Driver Channel A* define only synchronous method calls. Consequently, it was necessary to generate synchronous software connector fragments. Figure 8 illustrates the hardware components and

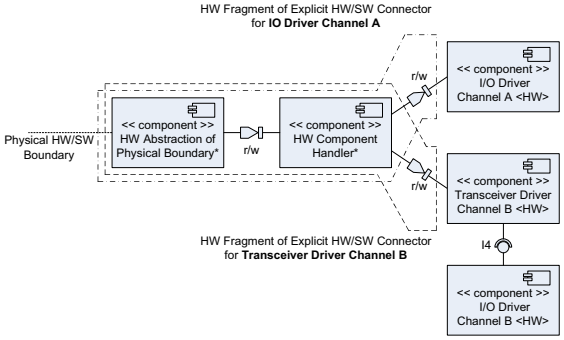

Figure 8. HW Connector Fragments ECU 1

the hardware connector fragments of ECU 1. The interface between *Transceiver Driver Channel B* and *I/O Driver Channel B* — marked as *I4* in Figure 8 — is a simple hardware/hardware interface and has been manually optimized to a setting and resetting of external FPGA pins. Automated generation and design of hardware/hardware component interfaces is also part of our research and has been already published in [9].

4.1. Transceiver Driver Component Benchmark

The main focus of benchmarking has been set to the hardware and software realization of the transceiver driver component. According to [10] we selected the performance metrics (i) *RAM Usage*, (ii) *ROM Usage*, (iii) *Worst Observed Execution TIME* (WOET) and (iv) *Average Observed Execution Time* (AOET) as significant properties for the hardware/software partitioning benchmark. Table 1 illustrates the resource usage of the transceiver driver components that have been implemented in software and hardware. The properties concerning the hardware implementation include the hardware realization of the FlexRay transceiver driver as well as the required hardware and software connector fragments. The benchmark of the transceiver driver component has been executed on an ALTERA EPXA4 device which has the following features:

- 166 MHz ARM922T 32 bit RISC CPU
- APEX20KE FPGA with 16640 logic elements and 26kByte internal RAM

According to Table 1, the hardware realization of the FlexRay transceiver driver together with the hardware connector fragment requires about 1.6 percent of the FPGA logic cells (258 of totally 16640 logic cells). The FPGA device on ECU 1 has already been equipped to embed a FlexRay communication controller. Thus the small hardware realization of a FlexRay transceiver driver could be

FlexRay Transceiver Driver	RAM [byte]	ROM [byte]	FPGA Logic Cells	FPGA RAM [byte]
SW Implementation	16	2101	-	-
HW Implementation	0	576	258	34

Table 1. Resource Usage of AUTOSAR FlexRay Transceiver Driver Component

FrTrcv_GetTransceiverWUReason	AOET [μs]	WOET [μs]
SW Implementation	210.0	303.6
HW Implementation	4.3	4.6

Table 2. Execution Times of AUTOSAR FlexRay Transceiver Driver Method FrTrcv_GetTransceiverWUReason

FrTrcv_GetVersion	AOET [μs]	WOET [μs]
SW Implementation	0.6	0.7
HW Implementation	2.8	2.9

Table 3. Execution Times of AUTOSAR FlexRay Transceiver Driver Method FrTrcv_GetVersion

integrated in the remaining unused logic cells with low effort. The advantage of the transceiver driver shift from software to hardware can be recognized if the most complex method of the transceiver driver is investigated in more detail. The method *FrTrcv_GetTransceiverWUReason* checks the status of a FlexRay transceiver driver device concerning physical bus wake-up reasons. The average and worst observed execution times (AOET and WOET) of Table 2 represent the duration of the synchronous (blocking) *FrTrcv_GetTransceiverWUReason* method call on the ALTERA EPXA4 platform. The tremendous advantage of the hardware implementation can be explained by investigation how the wake-up status information is read out from the transceiver device. The software implementation has to be scheduled to control the low-speed transceiver interface via I/O pins for the status readout. The alternative hardware implementation executes the readout cyclically in a second thread of execution and only provides the information during the method call. Nevertheless we have also investigated the communication overhead caused by the hardware/software connector fragments. It becomes visible if a simple method like *FrTrcv_GetVersion* is investigated in more detail, which provides information about the vendor and the version of the component. Table 3 illustrates the communication overhead of a hardware implementation in case of very simple method calls. The software and hardware connector fragments cause an execution time overhead that let the hardware implementation be slower than the software implementation. In general the communication overhead depends on the physical hardware/software interface, the clock frequencies of CPU and FPGA, and the number of parameters that have to be transferred. Thus a human engineer has to perform a trade-off for the hardware/software partitioning which can be supported by component measures and metrics as proposed in [10]. The proof-of-concept implementation verifies our concept of automated and thus rapid and faultless generation of all necessary interaction mechanisms between hardware and software components.

5. Conclusion

Today's complex embedded real-time system applications demand new approaches to fulfill system requirements concerning dependability, resource-efficiency, reusability, and real-time properties. Our approach extends the concept of component based software engineering (CBSE) to support automated and thus rapid and faultless hardware/software partitioning. A hardware component — realized in a field programmable gate array (FPGA) — together with the connector fragments in hardware and software replace a software component. Due to the well defined implementation methodology of a hardware component interface we provide for a tool that supports a flexible automated generation of hardware/software connectors. The final hardware/software partitioning decision is still based on human expert knowledge but our approach especially supports the modification and evaluation of different partitioning decisions. The proof-of-concept implementation demonstrates the functionality and advantages of our approach: Automated and rapid generation of hardware/software connectors as well as simple comparisons of different partitioning decisions.

6. Acknowledgment

This work has been partially funded by the FIT-IT [embedded systems initiative of the Austrian Federal Ministry of Transport, Innovation, and Technology] and managed by Eutema and the Austrian Research Agency FFG within project COMPASS[6] under contract 809444.

We would especially like to thank Martin Zauner from the University of Applied Sciences Technikum Vienna for supporting us in getting experiences with the ESTEREL design flow.

References

[1] AUTOSAR. *Automotive Open System Architecture*. http://www.autosar.org.

[2] AUTOSAR. *Specification of FlexRay Transceiver Driver V1.0.1*. http://www.autosar.org.

[3] Jakob Axelsson. Hardware/software codesign for automotive applications: Challenges of the architectural level. In *ISORC*, page 121. IEEE Computer Society, 2001.

[4] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[5] G. Berry. *The Foundations of Esterel*. MIT Press, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[6] COMPASS. *Component Based Automotive System Software*. http://www.infosys.tuwien.ac.at/compass.

[7] Martyn Edwards and Benjamin Fozard. Rapid prototyping of mixed hardware and software systems. In *DSD*, pages 118–125. IEEE Computer Society, 2002.

[8] FlexRay. *FlexRay Specification*. http://www.flexray.com.

[9] Wolfgang Forster and Eric Armengaud. A novel interconnection approach for globally asynchronous locally synchronous circuits. In *Austrochip 2007*, pages 107–114.

[10] Wolfgang Forster, Christof Kutschera, Dietmar Schreiner, and Karl M. Goschka. A unified benchmarking process for components in automotive embedded systems software. In *ISORC 2007*, pages 41–45. IEEE Computer Society, 2007.

[11] T. Führer, F. Hartwich, R. Hugel, and H. Weiler. FlexRay – The Communication System for Future Control Systems in Vehicles. In *Proceedings of the SAE 2003 World Congress & Exhibition*, Detroit, MI, USA, March 2003. Society of Automotive Engineers.

[12] Ying Guan, Yu-Ting Hung, and Rong-Guey Chang. Efficient hardware/software partitioning approach for embedded multiprocessor systems. In *International Symposium on VLSI Design, Automation and Test*. IEEE, 2006.

[13] P. Hansen. New s-class mercedes: Pioneering electronics. *The Hansen Report on Automotive Electronics*, 18(8):1–2, October 2005.

[14] H. Heinecke. AUTomotive Open System ARchitecture An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In *Proceedings of the Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, 2004.

[15] George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering*. Addison Wesley, 2001.

[16] IEEE Standards Association. *Open SystemC Language Reference Manual*. http://standards.ieee.org/getieee/1666/index.html.

[17] Information and Telecommunication Technology Center - University of Kansas. *Hthreads Innovative Computing Solutions*. http://www.ittc.ku.edu/hybridthreads/.

[18] Daniel Kroening and Natasha Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE*, pages 101–110. IEEE, 2005.

[19] David J. Lau, Orion Pritchard, and Philippe Molson. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. In *FCCM*, pages 45–56. IEEE Computer Society, 2006.

[20] OMG. *UML 2.0 Superstructure Specification*. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[21] Dietmar Schreiner and Karl M. Göschka. Explicit connectors in component based software engineering for distributed embedded systems. In *SOFSEM 2007: Theory and Practice of Computer Science, Proceedings*, volume 4362 / 2007 of *LNCS*, pages 923–934. LNCS, Springer, Jan 2007.

[22] Dietmar Schreiner and Karl M. Göschka. Synthesizing communication middleware from explicit connectors in component based distributed architectures. In *Proceedings of the 6th International Symposium on Software Composition (SC 2007)*, LNCS. Springer, 2007. to appear.

[23] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, January 1998.

[24] University of California, Berkeley. *POLIS - A Framework for Hardware-Software Co-Design of Embedded Systems*. http://embedded.eecs.berkeley.edu/research/hsc/.