Correct-by-Construction Generation of Device Drivers Based on RTL Testbenches

Nicola Bombieri Franco Fummi Graziano Pravadelli Sara Vinco
Dipartimento di Informatica
Università di Verona, Italy
Email: firstname.lastname@univr.it

***Abstract*—The generation of device drivers is a very time consuming and error prone activity. All the strategies proposed up to now to simplify this operation require a manual, even formal, specification of the device driver functionalities. In the system-level design, IP functionalities are tested by using testbenches, implemented to contain the communication protocols to correctly interact with the device. The aim of this paper is to present a methodology to automatically generate device drivers from the testbench of any RTL IP. The only manual step required is to tag the states corresponding to the different device functionalities. The Extended Finite State Machines (EFSMs) are then used to create a correct-by-construction two-level device driver: the lower level deals with architectural choices, while the higher one is derived from the EFSMs and it implements the communication protocols. The effectiveness of this methodology has been proved by applying it to a platform provided by STMicroelectronics.**

I. Introduction

The design of an embedded system is a very challenging task which involves the cooperation of different experts: system architects, SW developers, HW designers, verification engineers, etc [1]. Each of them operates on different views of the system starting from a very abstract informal specification and refining the model through different abstraction layers (see Figure 1). The system-level description is mapped onto an architecture to obtain a transactional-level model where communication is completely separated from computation. Finally, after partitioning, SW and HW parts follow different design flows.

Splitting the design tasks in HW and SW components introduces the need for an *interface* between the two parts that often is not specified in the initial requirements. This requires to implement a *communication bus* and *device drivers* to connect HW components, memory and programmable devices. Their purpose consists of hiding the HW to the SW layers by providing a set of functions to control the operations of the peripheral devices.

The design of such an interface is one of the most challenging tasks in the embedded system design flow [2]. First, the definition of device drivers is a very time consuming and error prone activity [3]. Moreover, embedded systems are designed for specific target hardware mechanisms. Therefore each system needs a device driver developed ex novo to control its devices. This makes reuse of already existing device drivers very hard for embedded systems. For these reasons, some works have been proposed which define strategies and frameworks to simplify the generation of device drivers [4], [5], [6], [7], [8], [9], [10], [11].

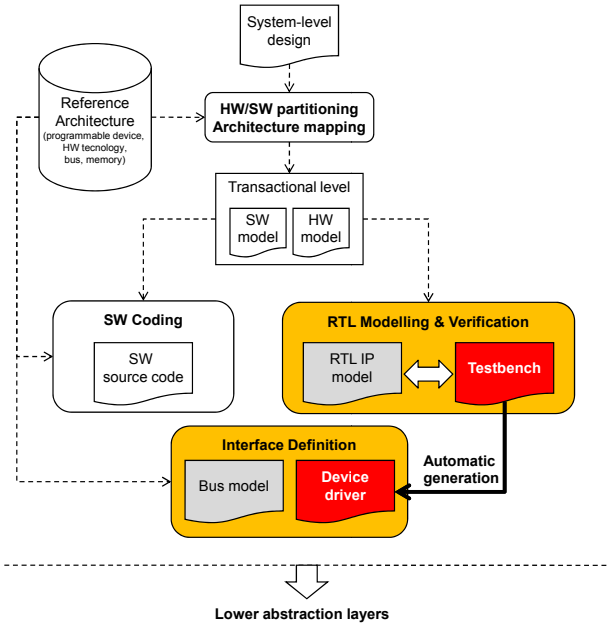

Fig. 1. Proposed device driver generation methodology in the system-level design flow.

In [4] the authors show how standard object oriented notations, like UML, can be used to simplify and support the development of specific pieces of code, like device drivers. [5] presents an integrated method to design a device and the corresponding device driver by using the SpecC language. However, both these papers do not propose a way for automatically generating the device driver code. An automatic generation method is proposed in [6], where the authors present an approach to HW/SW communication synthesis based on a regular language called ProGram. Synthesis is done from an architecture and implementation-dependent formal description of a device access protocol. A formal specification of the device driver functionality is used as starting point also in [7] and [8], where the authors derive the driver code from event driven state machine models. Finally, [9], [10], [11]

This work has been partially supported by the European projects VERTIGO FP6-2005-IST-5-033709 and COCONUT FP7-2007-IST-1-217069.

978-3-9810801-5-5/DATE09 © 2009 EDAA

describe frameworks that can be used to assist the design of device drivers specifically for Linux/Unix-based systems. To summarize, all previous works reported in the literature need a manual, even formal, specification of the device driver functionalities. Is this really necessary?

It is well known that system-level design uses *testbenches* to verify the RTL models refined from TLM specifications. Testbenches are pieces of code intended to stress and check all the features of the target device model to dynamically verify the correctness of the RTL models. To make verification effective, each testbench exactly implements the full set of operations that should be performed by the corresponding device driver to access the device functionality: polling and/or interrupt-based communication mechanisms, data exchange protocols, synchronization and register configuration phases, etc. Once that the RTL model of a device is ready to be synthesized, it is fair to assume that the corresponding RTL testbench is correct and reasonably complete. This paper assumes the completeness and correctness of the testbench as starting point of the methodology.

The idea lying underneath the proposed methodology is to *reuse* the information contained in these testbenches to *automatically generate* device drivers, without requiring error-prone and time consuming specification phases. A methodology for reusing information of the RTL communication protocol contained in the RTL testbenches has been originally proposed in [12] to automatically generate TLM transactors. In this paper we adopt the same idea but we show how such information can be exploited for automatically generating software device drivers.

The paper is organized as follows. Section II summarizes the main concepts related to CPU-device communication and device driver features. Section III presents the details of the proposed methodology. Section IV exemplifies the methodology by means of a complex test case provided by STMicroelectronics. Finally, Section V is devoted to concluding remarks and future works.

II. Device Driver Overview

Communication between software applications and hardware devices relies on two main concepts: the CPU-device communication and the device driver.

A. The CPU-Device communication

There are two standard ways to perform I/O operations between CPU and HW devices: memory-mapped I/O (MMIO) and port-mapped I/O (PMIO).

Adopting the MMIO technology, areas of CPU's addressable space are reserved for I/O as well as memory. Each I/O device monitors the CPU's address bus and responds to any CPU's access of device-assigned address space, connecting the data bus to a desirable device's hardware register.

On the other hand, the PMIO technology exploits a special set of CPU instructions for performing I/O. Thus, I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O.

In both cases, every peripheral device is controlled by writing and reading its registers. Most of the time a device has several registers accessed at consecutive addresses, either in the memory address space or in the I/O address space. At the hardware level, there is no conceptual difference between memory regions and I/O regions: both of them are accessed by asserting electrical signals on the address bus and control bus (i.e., the *read* and *write* signals) and by reading from or writing to the data bus.

The way used to access I/O memory (i.e., the device driver) depends on the computer architecture, bus, and device being used. However, the main mechanism used to communicate with devices is through MMIO.

B. The device driver

A real device usually offers more functionalities than synchronous `read` and `write` operations, such as initialization, configuration and various types of control operations. Each operation to/from the device (such as *send data* or *receive result*) must follow a precise protocol which implies data type, temporization and so on, to perform a correct CPU-device communication.

Each operation consists of a sequence of accesses (i.e., `write` and `read`) to the corresponding memory addresses. Such a kind of operations are usually supported via methods. As an example, the `ioctl` system call for Linux-based platforms offers a way to issue device-specific commands (like formatting a track of a floppy disk, which is neither a reading nor a writing operation). The `ioctl` commands are implemented by the kernel as device methods and called by the SW application by passing an identification number as system call parameter.

The main tedious and time-consuming task in writing a device driver is defining the device capabilities (i.e., the commands) that the driver will offer to SW programs.

III. Methodology Flow

The starting point is a system composed of the RTL code of the testbench and the RTL code of the target IP (see the leftmost side of Figure 2).

The RTL testbench is actually implemented to test the IP functionality. Thus, it performs ordered sequences of *read* and *write* operations: these sequences are executed in compliance with the IP communication protocol and they correspond to the several functions performed by the device driver. Thus, they are called *driver functions*.

The extended finite state machine (EFSM) model [13] has been adopted in this work to formally represent the behavior of the RTL testbench, and then the corresponding driver functions. This model allows us to conveniently manipulate the testbench for extracting the information required.

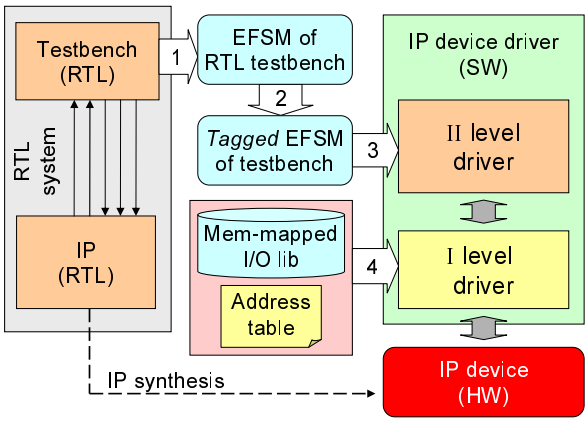

Fig. 2. Methodology overview

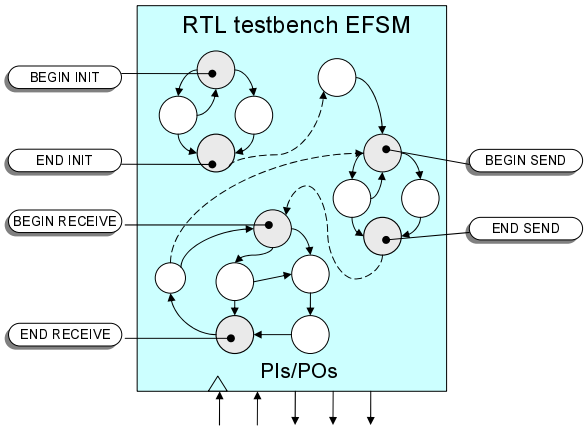

Fig. 3. Example of EFSM tagging

According to Figure 2, the proposed methodology is composed of four steps. In the *first step*,the EFSM representation of the testbench is extracted from the RTL description according to the strategy proposed in [14].

In the *second step*, a manual effort is required to identify which parts of the EFSM implement the sequence of statements written to access a specific IP functionality. Functionality identification is performed by *tagging* the EFSM states. This is the unique manual step of the methodology.

In the *third step*, the C code of the II level of the device driver is automatically generated from the tagged EFSM.

Finally, in the *fourth step*, the I level of the device driver is automatically generated. It implements the *memory-mapped I/O* approach, and thus it is common for every driver relying on this communication technology. An address table is settled to associate physical addresses with each primary input (PI) and primary output (PO) of the device.

A. Driver target architecture

The generation of the device driver relies on a two-level template (see the rightmost side of Figure 2).

The *II level device driver* implements the protocols to access the various IP functionalities. The methodology exploits information derived from the *driver functions*: in this way, this level is device specific and it implements a *command* (see Section II-B) for each driver functionality.

The *I level device driver* implements atomic operations used to access the device registers (such as `read` and `write`). These operations are invoked by the second level functions: the sequence of invocations forms the communication protocol. This level deals with many architectural choices (e.g., the addresses on which the device is mapped, interrupt handling, etc.). It is based on a template common to all devices.

If the system includes a bus, the same methodology can be applied to a RTL module of the bus itself. This will produce a two-level driver for the bus: other device drivers will communicate with the bus by invoking the functions of the second level bus driver.

B. Step 1: EFSM extraction

An EFSM is a transition system that allows a more compact representation of the design states with respect to the more traditional finite state machine (FSM). EFSMs represent the functionality of systems without requiring the explicit enumeration of all design states. In a conventional FSM, the transition is associated with a set of input boolean conditions and a set of output boolean functions. In an EFSM model, the transition can be expressed by an *if* statement, which could involve registers. If trigger conditions are all satisfied, the transition is fired, bringing the machine from the current state to the next state and performing the specified data.

The EFSM representation of the testbench can be automatically extracted from the corresponding RTL description as described in [14].

The EFSM model has been chosen for three main reasons. First of all, the EFSM is an extension of the traditional FSM model, which has been extensively used to model and verify protocols [15]. Secondly, thanks to the expressiveness of transitions, the driver code can be automatically generated starting from the EFSM representation. Finally, this model maintains the state explosion problem controlled by adding expressivity to transitions [16], thus allowing to extract EFSMs from any RTL code.

C. Step 2: EFSM tagging

After EFSM extraction, a preprocessing tagging stage on the EFSM is required to add information that cannot be automatically determined. This step represents the only necessary manual task and it is composed of two phases:

1) *Tagging of the EFSM subgraphs of the driver functionalities.* The device driver functionalities (e.g., send data, receive data, IP initialization, IP configuration, etc.) are identified in the EFSM of the testbench by tagging the initial and final states of the subgraph visited for testing the corresponding IP functionality. Figure 3 shows an example of EFSM tagging. The initial

DATA TABLE		
RTL port	Size (bit)	Type
Ips_module_en	1	IN
Ips_addr	14	IN
Ips_seq_access	1	IN
Ips_spv_access	1	IN
Ips_test_access	1	IN
Ips_rwb	1	IN
Ips_byte_en	4	IN
Ips_xfr_wait	1	OUT
Ips_wdata	32	IN
Ips_rdata	32	OUT

Fig. 4. Example of data table

ADDRESS TABLE				
RTL port	Size (bit)	Type	Address	Type
Ips_module_en	1	IN	DEV_ADDR_BASE	WONLY
Ips_addr	14	IN	DEV_ADDR_BASE + 1	WONLY
Ips_seq_access	1	IN	DEV_ADDR_BASE + 2	WONLY
Ips_spv_access	1	IN	DEV_ADDR_BASE + 3	WONLY
Ips_test_access	1	IN	DEV_ADDR_BASE + 4	WONLY
Ips_rwb	1	IN	DEV_ADDR_BASE + 5	WONLY
Ips_byte_en	4	IN	DEV_ADDR_BASE + 6	WONLY
Ips_xfr_wait	1	OUT	DEV_ADDR_BASE + 7	RONLY
Ips_wdata	32	IN	DEV_ADDR_BASE + 8	WONLY
Ips_rdata	32	OUT	DEV_ADDR_BASE + 9	RONLY

Fig. 5. Example of address table

state and the final state of EFSM subgraph performing an *INIT* operation are tagged with `BEGIN INIT` and `END INIT`. Similarly, `BEGIN SEND/END SEND` and `BEGIN RECEIVE/END RECEIVE` tags are used to identify the subgraphs of the EFSM performing data sending and data receiving. This provides the necessary support to automatically extract information about the RTL protocol encapsulated in the testbench.

2) *Identification of parameters of driver commands.* This phase determines the parameters passed by applications when invoking a device driver command. For each command, the parameter choice is strictly related to the EFSM subgraphs identified in the previous phase. For example, if the command performs a 32-bit data sending operation, the command parameter can be an unsigned 32-bit value, and the command implementation just implements the communication protocol to send such a data value. On the other hand, if an EFSM subgraph represents a loop where n sending operations are performed to transfer a 32-bit data at time, the command parameter will be a pointer to an array of unsigned 32-bit values, and the corresponding command will implement the loop and the communication protocol to send each one of the n data values.

D. Step 3: II level device driver

The II level of the device driver implements the protocols required to correctly use the functionalities of the corresponding IP core.

The automatic generation of the II level of the device driver consists of two main phases:

1) *Data table generation.* A data table is generated in which all the RTL IP ports are listed as well as the corresponding data size and type. Figure 4 shows an example of data table regarding an RTL IP implementing the IPS bus interface. The table is automatically generated by parsing the testbench code, in which the communication interface is implemented. The data table is necessary to generate the driver data structures that provide support for exchanging data between the device driver and the device. Moreover, the data table is analyzed and extended in order to associate a specific device address to each IP port, as explained in Section III-E.
2) *Device driver functionality generation.* The device driver functionalities correspond to the EFSM subgraphs of the testbench tagged during the first step of the methodology (Section III-B). The subgraphs of the EFSM are automatically elaborated to implement the device driver functionalities by exploiting the data table structures previously generated. For each command, the device driver sets each structure field following the corresponding protocol. In this context, if a field corresponds to a command parameter chosen in step 2 (Section III-C), that field is set with the formal parameter values passed by the application. On the other hand, data values for fields specific to the RTL interface (i.e., for handshaking sequences, byte enabling, burst cycles, etc.) are extracted from the RTL testbench. Thus, from the application point of view, the device functionalities are joined disregarding these details: since they are inherited from the testbench and transparently handled by the device driver.

E. Step 4: I level device driver

The final step of the proposed methodology consists of mapping the RTL ports listed in the data table generated during step 3 (Section III-D) to memory addresses, in order to implement a memory-mapped I/O approach. The RTL testbench accesses the RTL IP functionality by writing to PIs and reading from POs. Once the IP device has been synthesized, the driver performs the corresponding tasks by writing to and reading from physical addresses. Thus, each port of the RTL IP model is associated to a physical address assigned after synthesis and platform integration. In general, an address is associated to a device, while the PIs/POs of the device are identified by adding an offset to such an address. Figure 5 shows an example of address table generated by considering the data table presented in Figure 4 for a 32-bit

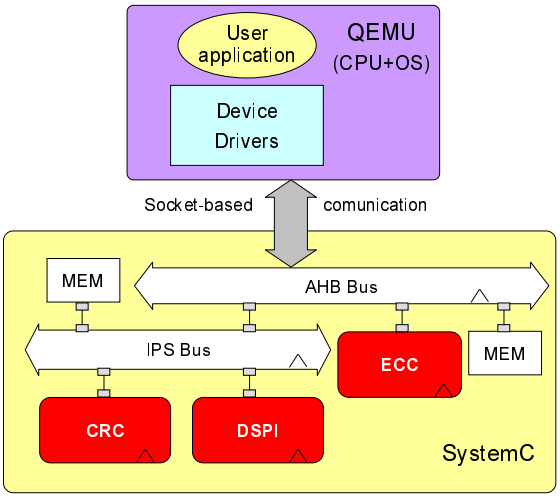

Fig. 6. The examined platform

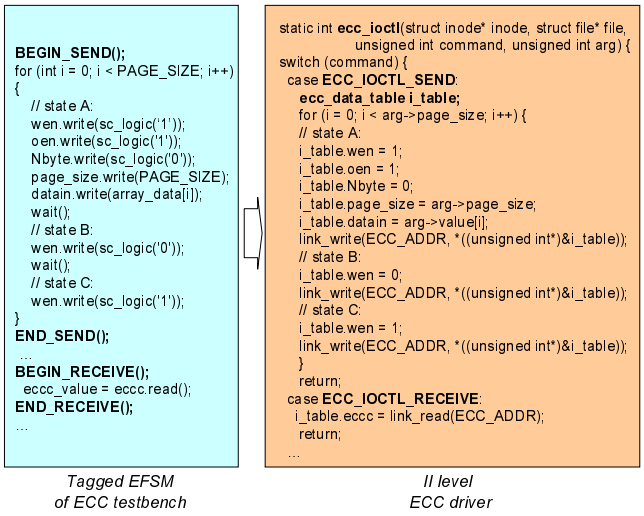

Fig. 7. Generation example of II level driver

technology. A device address is associated to each IP port and, generally, one address is associated to a port with size no larger than a word. If the port exceed the word size, it is associated to more than one address. On the contrary, one address is associated to a port smaller than a word (e.g., the 1-bit wide ports in Figure 4). Even if this algorithm may waste memory addresses, it ensures the generation process being automatic. Any optimization algorithm to manage addresses mapping can be finally developed and adopted.

It is important to note, that the proposed methodology is dependent on the CPU-device communication technology (i.e., memory-mapped I/O), while it is independent from the operating system on which the driver will be applied.

F. Methodology applicability and limitations

The proposed methodology strictly relies on the quality of the RTL device testbenches. As a consequence, the methodology has some limitations of applicability. Firstly, we assume that the starting point RTL testbenches are correct and reasonably complete. Nevertheless, correctness of the generated device drivers relies on correctness of the starting point RTL testbenches. Then, the EFSM models of the testbenches are extracted from the RTL testbenches. This is possible only if the testbenches comply with some simple writing rules. As an example, if several concurrent interactions were tested with a parallel submission of control signals by the testbench, the EFSM tagging would be quite hard to perform. A deeper analysis on dependency and quality of drivers with regards to the the RTL testbenches completeness and coding styles are included in our future work.

IV. Experimental Results

The four steps of the methodology described in Section III have been implemented on top of the HIFSuite [17]: this allows to automatically parse and manipulate the VHDL, Verilog or SystemC code of testbenches, in order to generate the C++ code of the device drivers.

The system specification and the RTL code of the whole platform (see Figure 6) have been provided by STMicroelectronics. Then, the SystemC model of the platform has been connected to a processor emulator (i.e. QEMU) to verify the behaviour of the device drivers. In this way, it is possible to simulate both the testbench (in SystemC) and the device driver (using QEMU) of each RTL module.

The methodology has been applied to automatically generate device drivers for the following devices:

1) *ECC*: an error correction code module;
2) *CRC*: a cyclic-redundancy checking module;
3) *DSPI*: a synchronous peripheral interface module.

Figure 7 shows, as an example, the II level driver generation for the ECC device, reporting the more meaningful parts of the code. The tagged EFSM of the ECC testbench is shown in the left side, where tags *BEGIN_SEND()/END_SEND()* and *BEGIN_RECEIVE()/END_RECEIVE()* mark the EFSM subgraphs of the corresponding *send* and *receive* functionality. In particular, the *send* functionality implements the communication protocol to send a page of 16-bit values towards ECC, while *receive* implements the protocol to read the corresponding error correction code of that page as result. Each statement of the testbench is translated to a corresponding C statement of the II level driver (right side of Figure7). The write and read operations made by the testbench on the PIs/POs of the IP, correspond to write and read operations made by the driver on the ECC addresses. At each state of the EFSM, a set of data values is sent/read on the PIs/POs, and the transitions between states are synchronized by the clock. On the other hand, the corresponding set of data values is written/read on the memory addresses by exploiting the *data_table* structure and the *link_write()/link_read()* functions. These *link* functions implement the common write and read operations on memory

Device	PIs (#)	POs (#)	Gates (#)	FF (#)	RTL testbench (loc)	Testbench EFSM		Driver functs. (#)	I level driver (loc)	II level driver (loc)
						#states	#trans.			
ECC	25	32	993	79	93	8	8	4	270	254
CRC	56	34	9,213	385	442	6	6	3	270	267
DSPI	25	21	1,335	132	462	6	6	3	270	292

TABLE I
EXPERIMENTAL RESULTS

addresses and synchronize the state transitions.

Table I reports the experimental results obtained by applying the proposed methodology to generate the device drivers for the system devices presented above, by choosing a target architecture composed with a CPU ARM926EJ-S and an O.S. Debian Linux 3.4.6-5. The characteristics of each device are reported in columns *PIs* (Primary Inputs), *POs* (Primary Outputs), *Gates*, and *FF* (Flip Flops). Column *RTL testbench* shows the number of lines of SystemC code of the testbenches which have been analyzed by the parser for extracting the driver information. Column *Testbench EFSM* shows the number of states and transitions of the EFSMs extracted from the RTL testbench, which model the driver functions. Column *Driver functs.* reports the number of driver functions available to the user application of each device. Finally, columns *I level driver* and *II level drivers* report the lines of generated C code of each driver.

For each device, few minutes of manual work have been spent for the preliminary step. Then, the automatic device driver generation has been instantaneously accomplished by our prototype tool. On the other hand, 5 person-days have been spent for implementing the equivalent three device drivers by hand.

V. CONCLUDING REMARKS AND FUTURE WORK

The paper addressed the problem of correct-by-construction automatic generation of device drivers. This has been obtained by reusing the testbenches developed to dynamically verify the RTL model of the devices. The methodology extracts the EFSM model of the testbench. Then, after a simple manual tagging phase, the driver functionality is derived from the EFSM. Even if the tagging manual phase can be considered a limitation, it is much easier and quicker than completely coding device drivers from scratch.

The methodology effectiveness and correctness have been shown by generating the device drivers of a complex industrial platform provided by STMicroelectronics.

Future work will deal with a comparative analysis between the drivers automatically generated and the drivers manually implemented, aiming at the performance improvement of the generated code. Future work will deal also with the extension of the methodology to address the generation of device drivers which are able to handle multiple-processor architectures, thus managing concurrent accesses to the device.

REFERENCES

[1] H. Yu, R. Domer, and D. Gajski, "Embedded software generation from system level design languages," in *Proc. of ASP-DAC, Asia and South Pacific*, 2004, pp. 463–468.

[2] *International Technology Roadmap for Semiconductors 2007 - Design*, http://www.itrs.net, 2007.

[3] H. Takada, "The recent status and future trends of embedded system development technology," *Journal of IPSJ*, vol. 42, no. 4, pp. 930–938, 2001.

[4] H. Sertic, F. Rus, and R. Rac, "Uml for real-time device driver development," in *Proc. of IEEE ConTel*, 2003, pp. 631–636.

[5] S. Honda and H. Takada, "Evaluation of applying specc to the integrated design method of device driver and device," in *Proc. of ACM/IEEE DATE*, 2003, pp. 138–143.

[6] M. O'Nils and A. Jantsch, "Device driver and dma controller synthesis from hw /sw communication protocol specifications," *Design Automation for Embedded Systems, Springer Netherlands*, vol. 6, no. 2, pp. 177–205, 2001.

[7] S. Wang, S. Malik, and R. A. Bergamaschi, "Modeling and integration of peripheral devices in embedded systems," in *Proc. of ACM/IEEE DATE*, 2003, pp. 136–141.

[8] S. Wang and S. Malik, "Synthesizing operating system based device drivers in embedded systems," in *Proc. of IEEE CODES+ISSS*, 2003, pp. 37–44.

[9] T. Katayama, K. Saisho, and A. Fukuda, "Prototype of the device driver generation system for unix-like operating systems," in *Proc. of IEEE Int. Symp. on Principles of Software Evolution*, 2001, pp. 302–310.

[10] Y.-T. Hsu, Y.-J. Wen, and S.-D. Wang, "Embedded hardware/software design and cosimulation using user mode linux and systemc," in *Proc. of IEEE ICPPW*, 2007, pp. 17–22.

[11] J. C. Park, Y. H. Choi, and T. ho Kim, "Domain specific code generation for linux device driver," in *Proc. of IEEE ICACT*, 2008, pp. 101–104.

[12] N. Bombieri, N. Deganello, and F. Fummi, "Integrating RTL IPs into TLM designs through automatic transactor generation," in *Proc. of ACM/IEEE DATE*, 2008, pp. 15–20.

[13] D. Lee and M. Yannakakis, "Online minimization of transition systems (extended abstract)," in *Proc. of ACM STOC*, 1992, pp. 264–274.

[14] K.-T. Cheng and A. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM TODAES*, vol. 1, no. 1, pp. 57–79, 1996.

[15] P. Chu and M.T.Liu, "Synthesizing protocol specifications from service specifications in FSM model," in *IEEE CNS*, 1988, pp. 173–182.

[16] F. Slomka, M. Dorfel, and R. Munzenberger, "Generating mixed hardware-software systems from SDL specifications," in *ACM/IEEE CODES+ISSS*, 2001, pp. 116–121.

[17] ESD Group, "HIFSuite," www.edalab.it/HIFSuite.