Semi-Automatic Generation of Device Drivers for Rapid Embedded Platform Development

Andrea Acquaviva, *Member, IEEE,* Nicola Bombieri, *Member, IEEE,* Franco Fummi, *Member, IEEE,* and Sara Vinco, *Member, IEEE*

***Abstract*—IP core integration into an embedded platform implies the implementation of a customized device driver complying with both the IP communication protocol and the CPU organization (single processor, SMP, AMP). Such a close dependence between driver and platform organization makes reuse of already existing device drivers very hard. Designers are forced to manually customize the driver code to any different organization of the target platform. This results in a very time-consuming and error-prone task. In this paper, we propose a methodology to semi-automatically generate customized device drivers, thus allowing a more rapid embedded platform development. The methodology exploits the testbench provided with the RTL IP module for extracting the formal model of the IP communication protocol. Then, a taxonomy of device drivers based on the CPU organization allows the system to determine the characteristics of the target platform and to obtain a template of the device driver code. This requires some manual support to identify the target architecture and to generate the desired device driver functionality. The template is used then to automatically generate drivers compliant with 1) the CPU organization, 2) the use in a simulated or in a real platform, 3) the interrupt support, 4) the operating system, 5) the I/O architecture, and 6) possible parallel execution. The proposed methodology has been successfully tested on a family of embedded platforms with different CPU organizations.**

***Index Terms*—Device driver design, embedded systems, hardware-software co-design, platform development.**

I. Introduction

CURRENT AND NEXT generation high-performance systems-on-chip (SoC) are characterized by multiple computational cores and a number of integrated peripherals, accelerators, and reconfigurable logic sections [1]–[3]. The design process of such heterogeneous multicore systems is increasingly complex and time consuming, as it requires the exploration of many alternatives to address the even more pushing performance requirements of applications and to handle heterogeneous use cases [4]. Such a component-level design exploration has a huge impact on the development of hardware-dependent software (HdS) [5], which includes device drivers.

Manuscript received August 2, 2012; revised February 11, 2013; accepted March 18, 2013. Date of current version August 16, 2013. This work was supported in part by the European Project TOUCHMORE under Grant FP7-ICT-2011-7-288166. This paper was recommended by Associate Editor S. Parameswaran.

A. Acquaviva is with the Department of Control and Computer Engineering, Politecnico di Torino, Turin 10121, Italy (e-mail: andrea.acquaviva@polito.it).

N. Bombieri, F. Fummi, and S. Vinco are with the Department of Computer Science, University of Verona, Verona 37100, Italy (e-mail: nicola.bombieri@univr.it; franco.fummi@univr.it; sara.vinco@univr.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2013.2257924

Design and customization of HdS and, in particular, of device drivers is key to reducing time-to-market and it entails two main aspects. From one side, the communication protocol between processing cores and devices depends on the device architecture and interface. On the other side, synchronization of accesses to a device depends on the number and type of processing cores that may have access to the device. A specific device can be accessed by multiple cores each running its own OS and device driver. On the other side, in a shared memory symmetric multiprocessor system (SMP) a single driver running from the shared memory has to arbitrate between different access requests possibly coming from the various cores.

The close dependence between driver and platform organization makes reuse of already existing device drivers very hard. Thus, designers have to manually customize the driver code to any different organization of the target platform, through a very time consuming and error-prone process. This limits the exploration of alternative platforms that is crucial for an effective selection of the target platform.

This paper proposes a semi-automatic device driver generation and customization methodology for rapid development of embedded platforms to support design space exploration during platform development.

Instead of requiring formal specification or manual implementation of the device functionalities, the proposed technique exploits information available during the device design and distributed together within the device itself. The last step before actual synthesis of a device is register transfer level (RTL) modeling. At RTL, the device is defined in terms of flow of signals (or transfer of data) between hardware registers and of logical operations performed on those signals. Such an implementation is called a RTL model of the device and it is provided with a testbench, used to stimulate and verify the device behavior. The methodology proposed in this article exploits this testbench to automatically generate the formal model of the HW/SW communication protocol. Such a formal model is then tagged by the user, to identify the provided functionalities.

The quality of the produced device driver will strictly depend on the starting testbench code. If the testbench does not activate all the device functionalities or it does provide only a subset of the available features, then the final device driver will be incomplete as well. Despite that, the methodology allows for the saving of design and verification time to write the device driver by exploiting the testbench. The claim is that manually tagging the testbench, and even making small modifications to it, is a reasonable effort compared to the manual implementation of the driver itself.

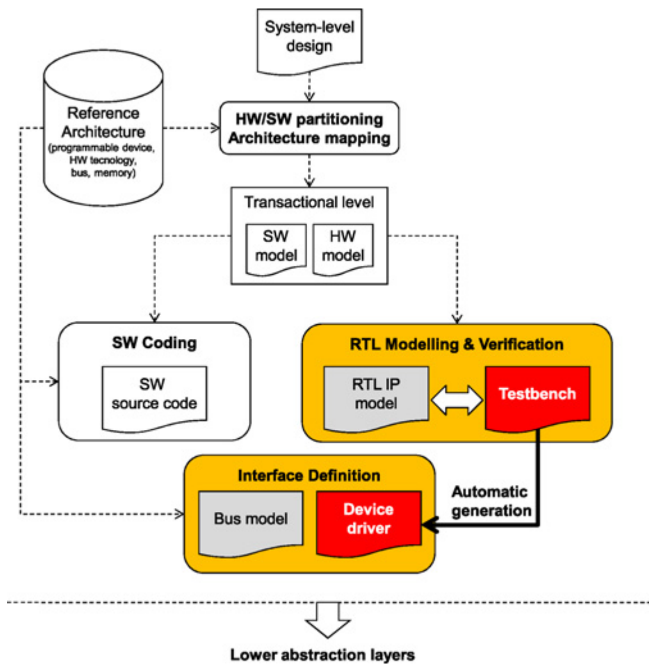


Fig. 1. Integration of the proposed methodology in a system-level design flow.

This article is a proof-of-concept that demonstrates that it is possible to save design and verification time to write device drivers by exploiting the testbench and avoiding formal specifications. In the current states, the proposed methodology addresses device drivers for embedded devices of medium/low complexity. Addressing more complex classes of devices, communication mechanisms, such as DMA or complex buffer management, or performance or timing requirements are interesting extensions of the proposed approach and part of our future work.

The article also focuses on supporting device driver generation for alternative CPU organizations. Thus, the generated code is customized to fulfill the requirements of the target platform, whose description is selected by the designer among a set of architectural alternatives. Synchronization primitives are added whenever needed by the specific CPU model and organization (either single processor, SMP, or AMP) of the target platform. Finally, the device driver customization methodology is fully compliant with real or simulated multicore platforms.

The customization methodology relies on manual specification of target platform and OS characteristics by the designer. However, the effort required is minimal and code generation is completely automated.

Fig. 1 shows how the proposed methodology can be integrated into a modern system-level design flow [6], in which different design groups operate on different system views. Starting from an abstract informal specification, the system model is refined through different abstraction layers. The system-level description is mapped onto an architecture to obtain a transactional-level model where communication is completely separated from computation. After partitioning, each HW device is refined through an RTL modeling and verification step, which in turn enables the automatic generation of the device driver with the methodology proposed in this article, and, finally, the device integration into the embedded platform.

The methodology has been implemented in a tool, D^2Gen, which provides a graphical interface for guiding the designer all along the generation flow, until code generation is accomplished. D^2Gen is built on top of HIFSuite [7] and it has been exploited to generate drivers for a set of benchmarks targeting a family of embedded platforms with different CPU organizations. Since generation is semi-automatic and the goal is to obtain drivers for embedded devices during the design space exploration, the generated device drivers are not highly optimized. However, D^2Gen allows for the obtainment of a fast implementation of the device drivers and thus to integrate the devices in the target platform. Furthermore, D^2Gen can generate device drivers for different target configurations, thus allowing design space exploration and the comparison of different alternatives while performing platform development. To the best of our knowledge, this is the first paper proposing a comprehensive device driver generation methodology targeting embedded platforms with various CPU organizations.

The article is organized as follows. Section II outlines related work about automatic device driver generation. Section III presents architectural scenarios, the related device driver taxonomy and an analysis of the most common RTL testbenches. Section V presents an overview of the driver generation methodology. Section VI reports the generation methodology in detail. Experimental results are reported in Section VII, whereas Section VIII concludes the paper.

II. Related Work

In [8] 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. [9] presents an integrated methodology to design a device and the corresponding 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 [10], 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 [11] and [12], where the authors derive the driver code from event driven state machine models. Also [13] uses an initial formal specification based on state machines to gain automatic device driver generation. The specification describes the functionality of the device driver, depicted as a set of events, the interface with the host CPU and the operating system requests handled by the device driver itself. [14]–[16] describe frameworks that can be used to assist the design of device drivers specifically for Linux/Unix-based systems. [17] supports device driver implementation by allowing automatic generation of low level code accessing the device. Formal specification of device ports and registers is necessary. Then, the designer can complete the device driver by using the generated low level code as access functions to the device. [5], [18] propose two techniques for HdS generation in general, which rely on manual specifications of the software architecture.

[19] automates reverse engineering of device drivers. Device driver code is synthesized, such that it implements the same communication protocol of the starting description, even if targeting a different operating system. As a drawback, an

original implementation of the device driver is still necessary and the quality of the final device driver will strictly depend on the original code.

All previous works require a manual and formal specification of the device driver functionalities. Furthermore, only a few of the works support platform configuration, since device drivers are generated for a specific architecture and, thus, exploration of alternative platforms is limited.

III. Methodology Starting Point

In this section we present the starting point of the generation methodology, the two-level structure used for the generated device drivers and an analysis of the key concepts that strongly affect the driver structure.

A. Device Driver Structure

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 write and read accesses 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 systems offers a way to issue device-specific commands (like formatting a memory, 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 parameters.

The proposed methodology relies on the two level template for device driver generation presented in [20] and consisting of a two-level structure [21].

1) The *high level driver* implements the communication protocol to correctly interact with the device.
2) The *low level driver* accesses device registers and deals with many architectural choices, such as address mapping and interrupt handling. The low level driver operations are invoked by the high level functions to implement the device communication protocol.

This structure is flexible and allows one to handle in a very efficient way the device communication protocol, together with concurrent access to shared resources and interrupt based synchronization.

The main task of the *high level driver* is implementing the communication protocol to correctly interact with the device. The protocol is a sequence of invocations of read and write functions of the low level driver. The methodology exploits information derived from the *driver functions* contained in the testbench. In this way, this level is device specific and it implements a *command* for each driver functionality. Interrupt detection and identification add management of interrupts generated by the device.

The *low level driver* deals with many architectural choices. Its main task is to provide low level communication with the device, with read and write operations on the device registers (when the platform supports real execution) or with invocations of precise functions provided to send data to the simulated platform (with co-simulation).

The most common way to perform I/O operations on a device is Memory Mapped I/O (MMIO). Adopting the MMIO technology, areas of a 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 hardware register. Thus, the low level device driver must handle MMIO communication with devices.

Finally, the low level device driver also handles interrupts generated from the communication infrastructure. In platforms with co-simulation, interrupts may arise to notify that the RTL level simulated devices have finished their computation and that data is available. Such interrupts do not interfere with normal OS operations but they are necessary to handle synchronization in co-simulation frameworks.

If the system includes a bus, the same methodology can be applied to a RTL module of the bus for generating a two-level bus driver. Other device drivers will communicate with the bus by invoking the functions of the high level bus driver.

Fig. 2 presents code segments of a user application, a high level device driver and two different implementations of the low level driver, the one communicating with a real device (left side of Fig. 2), the other communicating with a device simulated through co-simulation.

B. Device Driver Taxonomy

The code of a device driver is heavily affected by the characteristics of the target architecture where it will be executed, e.g., in terms of mutual exclusion resources or OS primitives. As a result, it is possible to build a taxonomy of the possible architecture configurations. Choosing one of the taxonomy options will provide the methodology with all the necessary information about the target architecture and thus it will allow to generate a device driver specific for the chosen architecture configuration.

Fig. 3 depicts the device driver taxonomy. In the Figure, each device class is represented by outlining (from top to bottom): number of running user programs, number of operating systems running, availability of both high level and low level device drivers, number of processors contained in the target architecture and number of devices. The circles represent the presence of mutual exclusion resources. The taxonomy is based on the *CPU organization* that is being used:

1) *Single processor* configurations are represented in the leftmost column of Fig. 3, where one operating system runs on a single processor. If single processor kernels are non-preemptive, they allow access to any kernel module to just one application at one time. This prevents race conditions and concurrent access to devices. On the other hand, preemptive kernels (e.g., real-time oriented kernels) require to adopt the mutual exclusion solutions to preserve the correct execution and to protect device status information.
2) *SMP systems* are multiprocessor computer architectures where two or more identical processors can connect to a single shared memory. This implies that all the processors access a single copy of the OS, even if applications run on different cores (as shown in the central column of Fig. 3). In this case, as in the case of preemptive kernels in single processor configuration, proper mutual exclusion mechanisms must be implemented.
3) *AMP systems* are multiprocessor computer architectures where two or more heterogeneous processors access

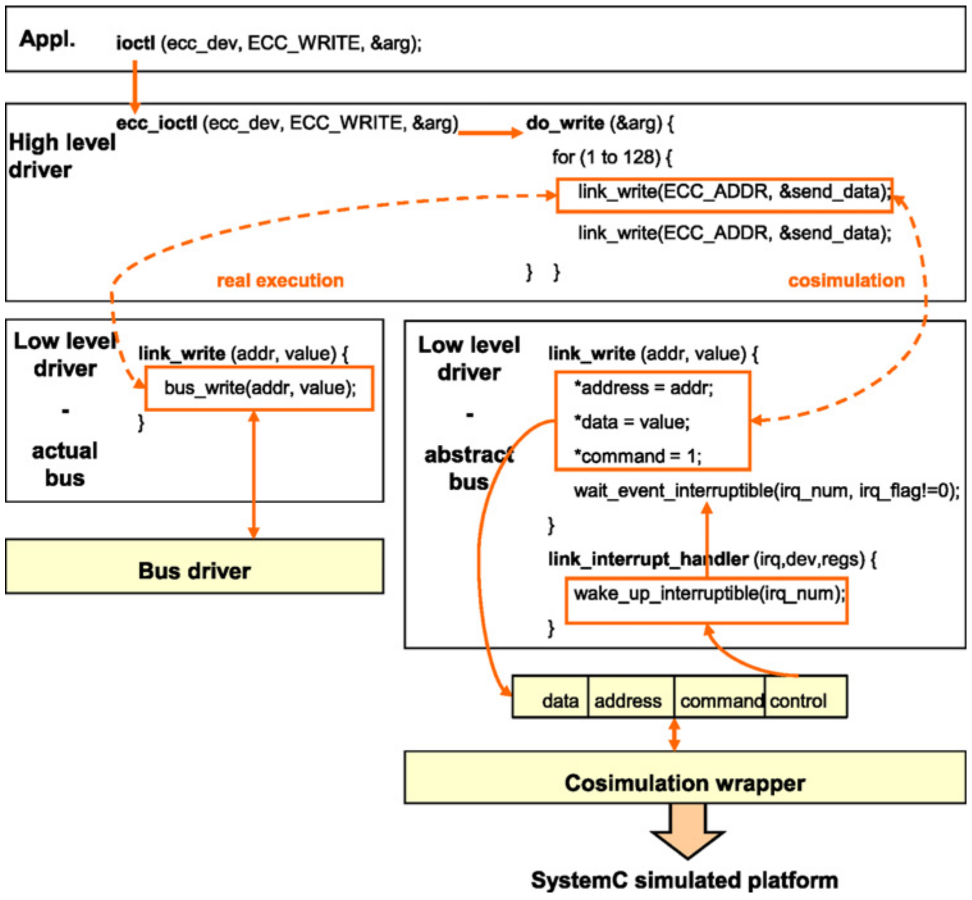

Appl.
ioctl (ecc_dev, ECC_WRITE, &arg);
High level driver
ecc_ioctl (ecc_dev, ECC_WRITE, &arg)
do_write (&arg) {
for (1 to 128) {
link_write(ECC_ADDR, &send_data);
link_write(ECC_ADDR, &send_data);
} }
real execution
cosimulation
Low level driver - actual bus
link_write (addr, value) {
bus_write(addr, value);
}
Bus driver
Low level driver - abstract bus
link_write (addr, value) {
*address = addr;
*data = value;
*command = 1;
wait_event_interruptible(irq_num, irq_flag!=0);
}
link_interrupt_handler (irq,dev,regs) {
wake_up_interruptible(irq_num);
}
data
address
command
control
Cosimulation wrapper
SystemC simulated platform

Fig. 2. Execution flow of a typical HW/SW interaction through device drivers. Two different implementations of the low level driver are proposed, the one communicating with a real device (left-hand side), the other communicating with a device simulated through co-simulation (right-hand side).

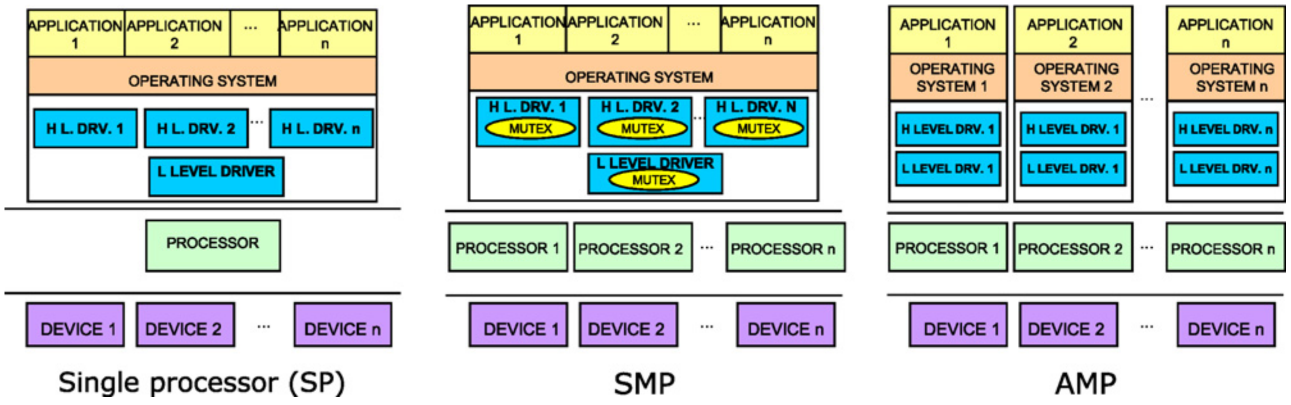


Fig. 3. Device driver taxonomy based on the CPU organization (single processor, SMP, or AMP).

the interconnect to communicate with a certain number of devices (as depicted in the rightmost column of Fig. 3). In most of the current industrially relevant architectures, either master-slave or symmetric, a single processing element runs the driver for a specific device. In case of master-slave architecture, it is the general purpose core playing this role, whereas in symmetric systems it could be one of the cores. In these cases the AMP configuration can be treated as the uniprocessor case.

The device driver generation methodology is also influenced by the *number of devices* that are foreseen in the platform and that use device drivers implemented with the same methodology. The low level device driver only deals with architectural details. Thus, it is enough to generate only one low level device driver per platform that will be shared by all devices connected to the same platform (multiple devices scenario). Some mutual exclusion resources must protect resources and operations that directly access the devices (e.g., spinlocks [22]). Else, if it is necessary to generate one low level driver per device, e.g., to allow an exclusive access to a certain device by one specific application, then the scenario is of type single device. The single device and multiple devices scenarios are depicted and compared in Fig. 4.

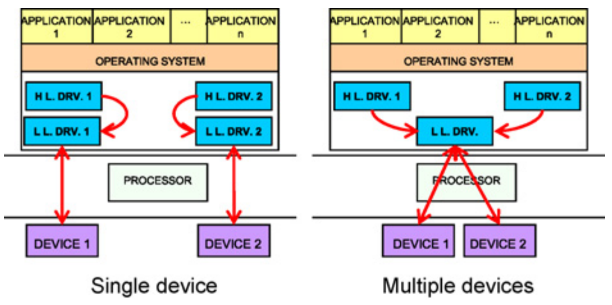


Fig. 4. Comparison of the single device (a) and multiple devices (b) scenarios.

C. Testbench Structure

Testbenches are a key feature in the design of complex devices. 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 [23]. The proposed methodology starting point is a set of testbenches, implementing the communication protocol necessary to activate all the device functionalities.

The testbench interacts with the target device through a set of read and write operations, compliant with the device communication protocol and corresponding to the functions performed by a device driver for this device. They are called *driver functions*. To make verification effective, each testbench must stimulate all the operations that can be performed by the device and implement all the possible interactions with the device functionality: polling and/or interrupt-based communication mechanisms, data exchange protocols, synchronization and register configuration phases, etc. Once the RTL model of a device is ready to be synthesized, the corresponding RTL testbench must be correct and reasonably complete.

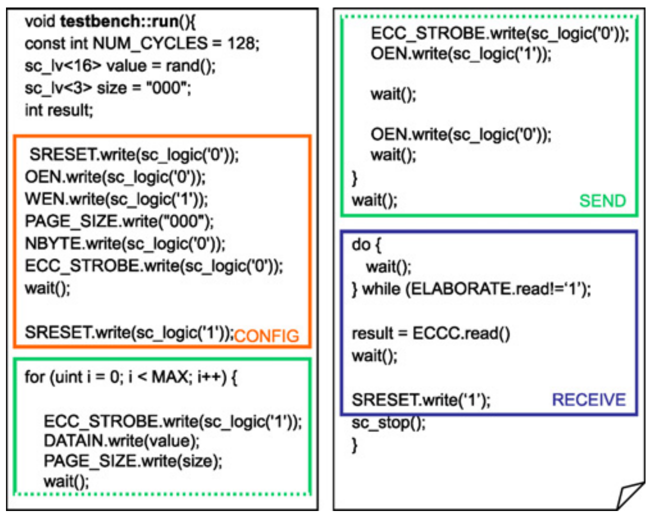
The core assumption of this paper is that a testbench (or a set of testbenches) is built to intensively test the functionality of a device. The completeness of the device driver will thus depend on the activation of all device functionalities by the testbench. This assumption is reasonable for well-designed devices, which are usually distributed with high-quality testbenches.

The proposed methodology assumes a RTL testbench describing all physical details (such as transactions, data length, ports, usage of blocking or non blocking operations, etc.), which are necessary for the driver to work correctly.

Fig. 5 provides an exhaustive example of testbench code. The testbench stimulates an error correction code (ECC) device. First of all the code performs a configuration phase. Then, a set of inputs are sent to the device and the final result is read. Fig. 5 also highlights the different functionalities activated by the testbench: configuration, send and receive. Such functionalities will be used to generate the corresponding device driver functions.

D. Co-Simulation Versus Real Execution

Tasks such as design exploration and functional validation might require the integration of the RTL device in the target platform before the device is synthesized or the target architecture is available. In these conditions, co-simulation is a suitable solution since it allows the interaction of a CPU with a simulated hardware platform (realized with HDL languages as VHDL or SystemC). As a result, co-simulation is a key solution to gain device driver validation before the


```
void testbench::run(){
const int NUM_CYCLES = 128;
sc_lv<16> value = rand();
sc_lv<3> size = "000";
int result;

 SRESET.write(sc_logic('0'));
OEN.write(sc_logic('0'));
WEN.write(sc_logic('1'));
PAGE_SIZE.write("000");
NBYTE.write(sc_logic('0'));
ECC_STROBE.write(sc_logic('0'));
wait();

SRESET.write(sc_logic('1'));CONFIG

for (uint i = 0; i < MAX; i++) {

   ECC_STROBE.write(sc_logic('1'));
   DATAIN.write(value);
   PAGE_SIZE.write(size);
   wait();
   ECC_STROBE.write(sc_logic('0'));
   OEN.write(sc_logic('1'));

   wait();

   OEN.write(sc_logic('0'));
   wait();
}
wait();                          SEND

do {
   wait();
} while (ELABORATE.read!='1');

result = ECCC.read()
wait();

SRESET.write('1');         RECEIVE
sc_stop();
}
```

Fig. 5. Example of testbench for an error correction code (ECC) device, where configuration, send and receive functionalities are enclosed in boxes.

target platform has been realized [24]. Indeed the device driver interacts with both the operating system services (and with the user applications, to check the compliance w.r.t. the architecture and the SW layers) and with a simulated version of the target devices (to check the compliance w.r.t. the device communication protocol).

Co-simulation frameworks must recreate all HW-SW communication mechanisms and provide support for the execution of HdS. However, integration of HdS in a co-simulation framework requires code modifications, to exploit the services provided by the framework and to correctly interact with the environment. As such, the device driver generation methodology must take into account the support of co-simulation frameworks.

The proposed methodology adopts the co-simulation presented in [24] and depicted in Fig. 6, since [24] allows the interaction of device drivers with the target OS and with real user applications, thanks to an Instruction Set Simulator (ISS), and with the devices simulated in SystemC. In this way, communication with the respective drivers can be tested even if the devices have not been synthesized yet. The framework supports all mechanisms that allow HW/SW communication via socket. Interrupts are used to detect when the simulated platform has completed the application execution. Such interrupts do not impact on the OS execution, as the device driver is suspended and the registered interrupt number is chosen such that it is a non shared one. This approach allows designers to test correctness and performance of the generated device drivers before the HW design flow has finished.

The adoption of this co-simulation approach impacts on the communication between the device driver and the target device. As an example, the low level driver must use the read and write functions provided by co-simulation instead of directly writing to the device registers or using I/O operations. As a result, the proposed methodology supports co-simulation to allow validation of the generated device driver and to test its interaction with the device before the target final platform is ready.

IV. EFSM Extraction from RTL Code

An EFSM is a transition system that allows a compact representation of the design states with respect to the more

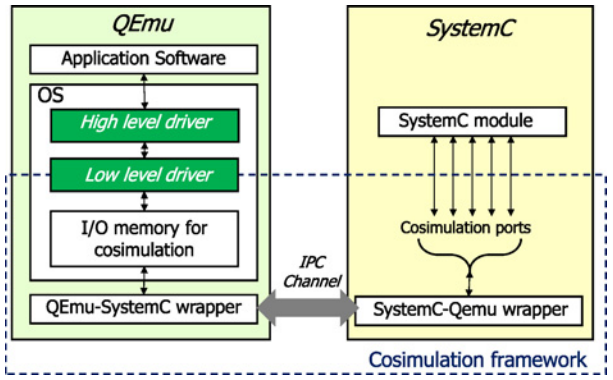


Fig. 6. Co-simulation approach proposed in [24].

```
1: $SG$ = get_synchronization_graph($tb\_file$);
2: $I$ = get_alphabet($IN$, $tb\_file$);
3: $O$ = get_alphabet($OUT$, $tb\_file$);
4: $D$ = get_alphabet($REG$, $tb\_file$);
5: for each vertex $v_i \in SG$ do
6:     $s_i$ = create_new_state($v_i$);
7:     $S$.add($s_i$);
8: end for
9: for each state $s_i \in S$ do
10:     for each state $s_j \in S$ such that $(v_i, v_j) \in SG$ do
11:         $T$.add($s_i$, $s_j$);
12:         $En$.add($s_i$, $s_j$, $SG$.get_conditions($v_i$, $v_j$));
13:         $Up$.add($s_i$, $s_j$, $SG$.get_instructions($v_i$, $v_j$));
14:     end for
15: end for
```

Fig. 7. EFSM extraction algorithm.

traditional finite state machine (FSM) [25]. The EFSM model partially mitigates the state explosion problem by adding expressivity to transitions and thus avoiding explicit enumeration of all design states [26].

Definition 1: An extended finite state machine (EFSM) is defined as a tuple $M = \langle S, I, O, D, En, Upd, T\rangle$ where:

1) S is a finite set of states;
2) $I = I_1 \times \ldots I_m$ is a finite input alphabet, where I_i is the alphabet of the i-th input;
3) $O = O_1 \times \ldots O_l$ is a finite output alphabet, where O_i is the alphabet of the i-th output;
4) $D = D_1 \times \ldots \times D_n$ is a finite register alphabet, where D_i is the alphabet of the i-th register;
5) T is the *transition relation*, that is a binary relation over states, $T \subseteq S \times S$;
6) En maps every transition to a boolean function named *enabling function*, $En : T \rightarrow (I \times D \rightarrow \{0, 1\})$;
7) Upd maps every transition to a register updating function named *update function*, $Upd : T \rightarrow (I \times D \rightarrow D \times O)$.

The rest of this section shows how the EFSM representation of a generic RTL implementation can be automatically extracted. The algorithm assumes that the testbench implementation contains only one process. Then, concatenation of EFSMs is defined, to extended the methodology to implementations with multiple processes and/or multiple testbenches.

Definition 2: A *synchronization point* is defined as a point in the execution of a HDL process where synchronization with the scheduler and with other processes is necessary. The synchronization points are:

1) explicitly enumerated states;
2) synchronization constructs contained by the code, such as invocations to the `wait` primitive in SystemC or VHDL.

Definition 3: A *synchronization graph SG* is a Control Flow Graph (CFG) where vertexes v are synchronization points and an edge (v_i, v_j) exists when v_j is one of the possible next states defined for v_i. Thus, *SG* is used to keep track of the relationship between synchronization points.

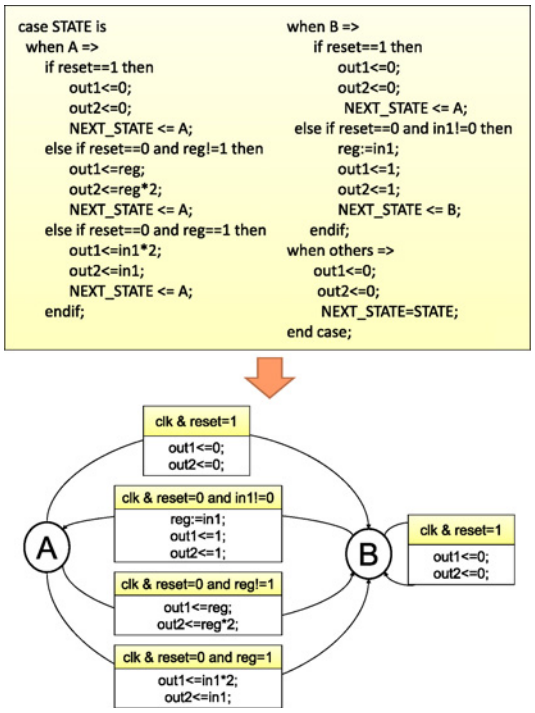
An EFSM representation is automatically extracted from a generic RTL implementation by applying the algorithm shown in Fig. 7. Given a RTL implementation $\mathcal{I}$ containing one process *P* and the synchronization point CFG $SG_{\mathcal{I}}$ built for $\mathcal{I}$, the EFSM $M = \langle S, I, O, D, En, Upd, T\rangle$ representing $\mathcal{I}$ is built as follows:

1) S is defined by one state s_i for each synchronization point v_i in $SG_{\mathcal{I}}$ (line 1 and lines 5-7);
2) I is defined by the set of alphabets for the inputs of $\mathcal{I}$, i.e., input ports of $\mathcal{I}$, all signals declared by $\mathcal{I}$ that are read by *P*, and all shared variables (line 2);[1]
3) O is defined by the alphabets for the outputs of $\mathcal{I}$, i.e., output ports of $\mathcal{I}$, all signals declared by $\mathcal{I}$ that are written by *P*, and all shared variables (line 3);
4) D is defined as the alphabets for the registers of $\mathcal{I}$ (line 4). Registers are a subset of registers and signals defined by $\mathcal{I}$. Signals and registers can be wires, i.e., used to connect different units, or registers, i.e., used to preserve a certain value across clock cycles. The latter are identified by synthesis patterns and implemented as memory elements (e.g., latches). Thus, synthesis patterns can be applied to identify the EFSM registers and their alphabet;
5) T is defined by the set of couples of states (s_i, s_j) such that $SG_{\mathcal{I}}$ contains an edge (v_i, v_j) between the synchronization point v_i, corresponding to s_i, and the synchronization point v_j, corresponding to s_j (lines 9-11). As a result, if T contains the couple (s_i, s_j), then s_j is one of the possible next states of s_i;
6) En is defined by the set of conditions that activate each transition $(s_i, s_j) \in T$ (line 12). The condition is expressed as a logical statement and of the following predicates:
    a) activation of the *sensitivity list*[2];
    b) *branch conditions* that influence the determination of the next state;
    c) *wait conditions*. If the node s_i corresponds to a `wait` synchronization point v_i in the $SG_{\mathcal{I}}$, then the conditions specified by the `wait` construct on signals must hold.

    As a result, the transition (s_i, s_j) can be activated only if, in $\mathcal{I}$, the sensitivity list fired an event, the `wait` conditions are verified and the branch conditions for reaching s_j as next state are satisfied.
7) Given (s_i, s_j) and the corresponding synchronization points (v_i, v_j) in the $SG_{\mathcal{I}}$, the Upd function is defined by all the instructions performed between the synchronization points v_i and v_j in $\mathcal{I}$.

[1]Note that the usage of shared variables is allowed in both VHDL and SystemC.

[2]This is built as a logic *OR* on the `'event` attribute of signals in VHDL-like notation. The same behavior can be easily represented in SystemC and Verilog.



```
case STATE is
  when A =>
    if reset==1 then
        out1<=0;
        out2<=0;
        NEXT_STATE <= A;
    else if reset==0 and reg!=1 then
        out1<=reg;
        out2<=reg*2;
        NEXT_STATE <= A;
    else if reset==0 and reg==1 then
        out1<=in1*2;
        out2<=in1;
        NEXT_STATE <= A;
    endif;
  when B =>
    if reset==1 then
        out1<=0;
        out2<=0;
        NEXT_STATE <= A;
    else if reset==0 and in1!=0 then
        reg:=in1;
        out1<=1;
        out2<=1;
        NEXT_STATE <= B;
    endif;
  when others =>
    out1<=0;
    out2<=0;
    NEXT_STATE=STATE;
end case;
```

Fig. 8. Example of EFSM generation starting from a HDL testbench.

Fig. 8 shows an example of EFSM generated from RTL code. The size of the generated EFSM is strictly related to the (finite) number of explicit states contained in the starting implementation. It is also easy to note that none of the performed operations is exponential in the number of analyzed statements.

In order to extend the methodology to multiple processes and testbenches, it is necessary to provide some definitions over the states of an EFSM $M = \langle S, I, O, D, En, Upd, T \rangle$.

Definition 4: An *initial state* for the EFSM M is a state that has no incoming edges. State s is an initial state iff. $\forall$ $(s_1, s_2) \in T$ $s \neq s_2$. Each EFSM has one only initial state, represented as $initial_M$.

Definition 5: A *final state* for the EFSM M is a state that has no outgoing edges. State s is a final state iff. $\forall$ $(s_1, s_2) \in T$ $s \neq s_1$. The set of final states of the EFSM M is represented as $finals_M$.

When more testbenches are provided for a single RTL design, they are built for stimulating different operations provided by the device independently. As such, the EFSMs extracted from each testbench process must be activated one after another, so that they do not affect each other's evolution. In this way, it is necessary to build a *concatenation of EFSMs*, rather than their composition (that may lead to state explosion [27]). Given two EFSMs M_1 and M_2, M_1 is put after M_2, with an edge connecting the final states of M_1 to the initial state of M_2. The enabling conditions of such edges are always satisfied and the update functions do not modify the EFSM state. An example of result of composition is shows in Fig. 11, where dashed arrows are the edges added by composition to connect the starting independent EFSMs.

The concatenation $M = (M_1, M_2)$ of two EFSMs $M_1 = \langle S_1, I_1, O_1, D_1, En_1, Upd_1, T_1 \rangle$ and $M_2 = \langle S_2, I_2, O_2, D_2, En_2, Upd_2, T_2 \rangle$ is thus defined as follows. All the constitutive elements of the EFSM M are built as the union of the corresponding elements of M_1 and M_2 (e.g., $S = S_1 \cup S_2$). Furthermore, some elements are enriched:

1) $T = T_1 \cup T_2 \cup \{\forall s \in final_{M_1}, (s, initial_{M_2})\}$, i.e., an edge is added between each final edge of M_1 and the initial state of M_2;
2) $En = En_1 \cup En_2 \cup \{\forall s \in final_{M_1}, (s, initial_{M_2}, \texttt{true})\}$, i.e., the edges added between each final edge of M_1 and the initial state of M_2 have an enabling function that always evaluates to true;
3) $Upd = Upd_1 \cup Upd_2 \cup \{\forall s \in final_{M_1}, (s, initial_{M_2}, \texttt{nop})\}$, i.e., the edges added between each final edge of M_1 and the initial state of M_2 have a simple update function that does not modify the state of the EFSM.

It is important to note that neither states nor transitions grow exponentially. States are exactly the union of states of M_1 and M_2. Transitions are built as the union of transitions of M_1 and M_2, increased only with a linear number of concatenation transitions (one per final state of M_1).

V. Methodology Overview

The proposed methodology combines automatic extraction of the communication protocol from the testbench with a few simple manual steps related to the CPU organization. In this way, the generated device driver follows the communication protocol and at the same time it responds to needs and problems that arise from the specific CPU organization.

The starting point of the methodology is a system composed of a RTL device and the related testbench (see top of Fig. 9). The proposed methodology is composed of five steps, that will be described in depth in Section VI. In the *first step*, the EFSM model of the testbench is extracted from its RTL description. Then, the methodology scans the EFSM to detect conditions that might be handled with interrupts. In the *second step* a manual effort is required to identify the parts of the EFSM that correspond to each driver function and interrupts. The target architecture characteristics are defined in the *third step*, to create a template for both the low level and the high level device drivers. In the *fourth step*, the C code of the high level device driver is automatically generated from the tagged EFSM and from the template generated in the third step. The architectural choices made in the previous steps might add interrupt handling routines or mutual exclusion resources implied by the chosen scenario. Finally, in the *fifth step*, the low level device driver is automatically generated by using the template produced by the third step to handle mutual exclusion, interrupts and communication with the device.

The few manual steps required create a very versatile methodology, that responds to the needs of a large set of architectures and architectural choices. The effort required is largely rewarded by the flexibility of the results obtained.

VI. Methodology in Depth

This section deepens the methodology steps shown in Fig. 9, with references to the technological issues analyzed in the previous section. All the examples and figures show the application of the methodology to the ECC device, whose testbench is depicted in Fig. 5.

A. First Step: EFSM Extraction and Interrupt Discovery

The *first step* provides a deep analysis of the starting code, divided into two major activities: extraction of the EFSM

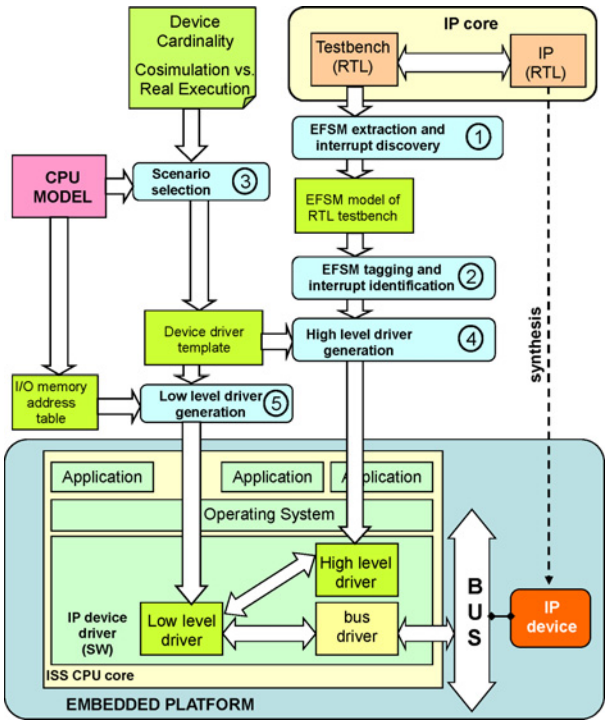
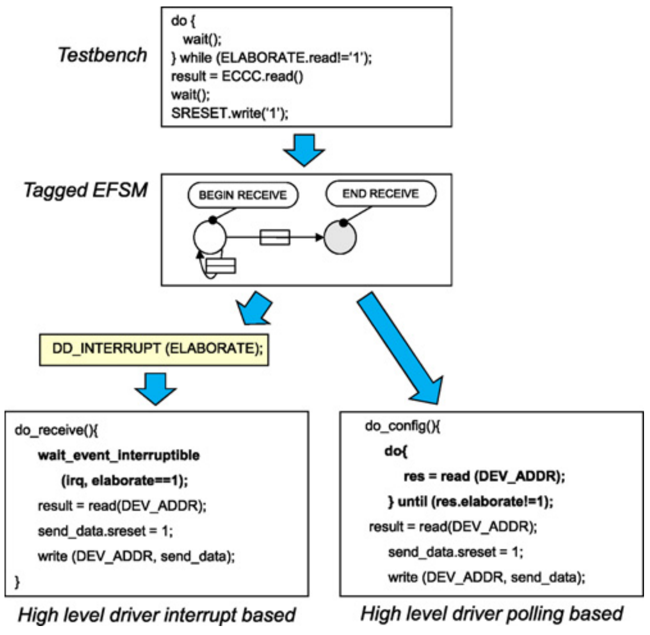


Fig. 9. Methodology overview.


Testbench
Tagged EFSM
BEGIN RECEIVE
END RECEIVE
DD_INTERRUPT (ELABORATE);
High level driver interrupt based
High level driver polling based

```
do {
  wait();
} while (ELABORATE.read!='1');
result = ECCC.read()
wait();
SRESET.write('1');
```

```
do_receive(){
    wait_event_interruptible
        (irq, elaborate==1);
    result = read(DEV_ADDR);
    send_data.sreset = 1;
    write (DEV_ADDR, send_data);
}
```

```
do_config(){
    do{
        res = read (DEV_ADDR);
    } until (res.elaborate!=1);
  result = read(DEV_ADDR);
    send_data.sreset = 1;
    write (DEV_ADDR, send_data);
```

Fig. 10. Effect of interrupt discovery on the generated code. The loop on the ELABORATE signal is handled with an interrupt on the left hand side, whereas polling is adopted on the right hand side.

model that represents the device communication protocol, and identification of interrupt-like behaviors.

1) *EFSM Extraction:* The EFSM representation of the testbench is automatically extracted from the corresponding RTL description with the algorithm proposed in Section IV. It is important to note that, in case more EFSMs are extracted from the testbench, such EFSMs can be concatenated in a new EFSM. As a result, the follow of this paper will refer to a single EFSM.

2) *Interrupt Discovery:* RTL testbenches are made of a sequence of read and write operations on RTL device ports. Testbenches control any communication with the device. Indeed, the device can not actively send a signal and raise an interrupt. On the contrary, the testbench may repeatedly read a device port until a certain condition is met, thus implying a polling communication style with the device. The result is that, in any testbench, interrupts actively sent from RTL devices become testbench operations. The testbench will repeatedly check the RTL ports until a certain condition happens, causing a hardware interrupt. Once that the testbench is turned into a synchronous EFSM, the methodology may identify interrupt-like conditions as self loops in the extracted EFSM (Fig. 10). Interrupt-based approaches are often more efficient than polling.

These considerations have effects on the device driver generation methodology. The EFSM is extracted from the testbench. Then, the methodology scans the EFSM looking for self loops and these conditions are pointed out to the user as possible interrupts. Both EFSM extraction and interrupt discovery are automated. In the second step (Section VI-B) of the methodology, the user will decide which self loops are real interrupt by using a special directive [`DD_INTERRUPT (accepted_interrupt_name)`]. Fig. 10 compares the device driver code generated by using interrupts (left side) or polling (right side).

Whenever a self loop is accepted as an *interrupt*, the methodology will substitute the state with an OS macro that suspends the execution until the corresponding interrupt is received (e.g., the `wait_event_interruptible` primitive in Linux OS). The interrupt handler function will be handled by the high level driver and it will just let the driver resume the execution of the communication protocol (e.g., with the `wake_up_interruptible` function in Linux OS).

All the self loops that are not accepted as interrupts will be handled with *polling*. In the resulting device driver they will be implemented with a while loop containing a read operation on the device registers, repeated until the guard condition is verified.

This mechanism allows designers to manage communication with the device in an efficient way and to handle hardware interrupts correctly. While requiring some manual specification by the designer, this improves correctness and effectiveness of the final code.

B. Second Step: EFSM Tagging and Interrupt Identification

After EFSM extraction, a preprocessing tagging stage on the EFSM is required. In detail, the designer is required to add information that can not be automatically determined, since it is strictly dependent on the specific device and on the desired behavior of the final device driver code.

1) *Tagging of the EFSM Subgraphs of the Driver Functionalities:* Device driver functionalities, such as send data, receive data, initialization and configuration, must be identified in the EFSM of the testbench by tagging the initial and final states of the subgraph visited for testing the corresponding device functionality. The EFSM extracted during the first step is shown to the designer, who can label the EFSM states as part of a certain functionality.

Fig. 11 shows an example of EFSM tagging applied to the testbench of Fig. 5. The initial state and the final state of EFSM subgraph performing a *CONFIG* operation are tagged

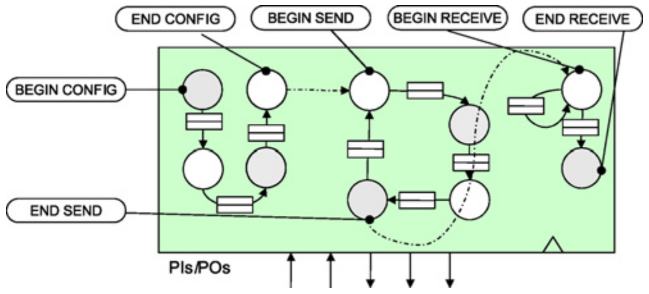


Fig. 11. Example of tagged EFSM obtained from the testbench in Fig. 5.

with BEGIN CONFIG and END CONFIG. 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 step provides the necessary support to automatically extract information about the RTL protocol encapsulated in the testbench.

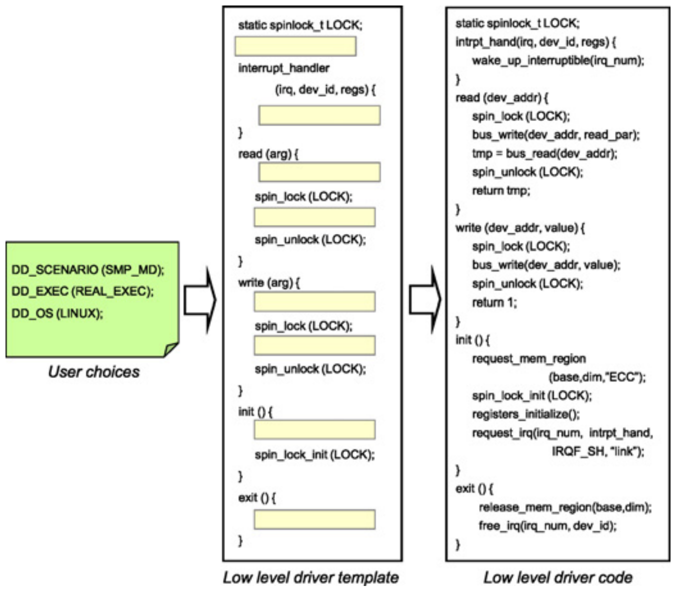
2) *Identification of Parameters of Driver Commands:* In this phase, the designer determines the parameters passed by applications when invoking a device driver command. Indeed, the data used to stimulate the device may be hard coded or chosen randomly. Some of the inputs may be configuration inputs that remain the same even in the actual driver. On the other hand, other inputs may depend on user defined data, e.g., inputs set by the application. To allow this level of flexibility, one can declare that certain variables or data passed in input should be considered parameters of the device driver functions, to allow passage of data from the application. This allows the device driver to take inputs that differ from the ones used in the starting testbench.

To perform this step, the designer is required to label variable declarations in the graphical representation of the testbench EFSM. Then, the methodology will consider all instances of the labeled variables as accesses to parameters of the driver functionalities. 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 designer can choose as command parameter the unsigned 32-bit value. On the other hand, if an EFSM subgraph represents a loop where n sending operations are performed transferring 32-bits of data at a time, the designer may identify as parameter the pointer to an array of unsigned 32-bit values, thus forcing the corresponding command to implement the loop and the communication protocol to send each one of the n data values.

3) *Interrupt Identification:* The user must consider the interrupt-like conditions pointed out in the previous step and decide which ones must be handled as real interrupts by using the DD_INTERRUPT keyword, associated to the interrupt identifier (Fig. 10). The consequences of this choice are described in Section VI-A2 and have effects on the high level driver content (Section VI-D).

C. Third Step: Architectural Specification

The third step is a crucial step in the methodology because the target platform characteristics are defined. This requires an interaction with the designer, that must specify the following information: target scenario, type of platform (i.e., whether the device driver will be executed in a cosimulated platform or in a real one) and device class. These choices are used by the methodology to generate a precise *template* for both the low level and high level drivers, containing (whenever needed) interrupt handling resources, ioctl functions, mutual exclusion resources and co-simulation structures (Fig. 12). Such templates can be automatically generated and they constitute a skeleton of the device driver code. Then, the template must be integrated with the device specific code and information extracted during the second step. The templates will be used in the next steps (Section VI-D and VI-E) for the device driver generation.



```
DD_SCENARIO (SMP_MD);
DD_EXEC (REAL_EXEC);
DD_OS (LINUX);
```

```
static spinlock_t LOCK;

interrupt_handler
        (irq, dev_id, regs) {

}
read (arg) {

    spin_lock (LOCK);

    spin_unlock (LOCK);
}
write (arg) {

    spin_lock (LOCK);

    spin_unlock (LOCK);
}
init () {

    spin_lock_init (LOCK);
}
exit () {

}
```

```
static spinlock_t LOCK;
intrpt_hand(irq, dev_id, regs) {
    wake_up_interruptible(irq_num);
}
read (dev_addr) {
    spin_lock (LOCK);
    bus_write(dev_addr, read_par);
    tmp = bus_read(dev_addr);
    spin_unlock (LOCK);
    return tmp;
}
write (dev_addr, value) {
    spin_lock (LOCK);
    bus_write(dev_addr, value);
    spin_unlock (LOCK);
    return 1;
}
init () {
    request_mem_region
                (base,dim,"ECC");
    spin_lock_init (LOCK);
    registers_initialize();
    request_irq(irq_num, intrpt_hand,
                IRQF_SH, "link");
}
exit () {
     release_mem_region(base,dim);
     free_irq(irq_num, dev_id);
}
```

Fig. 12. Example of specifications and corresponding low level driver template for a SMP architecture with real execution of the synthesized device.

1) *Target Scenario:* This parameter describes the platform characteristics, chosen from the taxonomy described in Section III. The choice has effects on the handling of concurrency and mutual exclusion, as well as on preemption of the device driver code. For example, scenarios that imply concurrency need to handle mutual exclusion with spinlock resources.

The designer is required to specify also other architectural information, such as the *OS* that will run on the target platform, as this has effects on the primitives and the mutual exclusion resources used, and the chosen endianness.

2) *Type of Platform:* The RTL device can be integrated in the target platform as a real device (after a synthesis process) or at RTL level in a co-simulation environment where a real system interacts with simulated devices. The device driver generation methodology must support both alternatives. During the third step (Section VI-C) the user can decide between a real-execution approach and a co-simulated approach. There are three possible scenarios, as shown in Fig. 13:

1) *Abstracted bus*. The target platform is realized at transaction level [28] with a non cycle-accurate bus. Co-simulation is between the low level driver and the simulated platform. Thus, the low level driver must use the function provided by the co-simulation environment to communicate with the device. This implies that the methodology must generate an appropriate low level driver.
2) *Actual bus* and *co-simulated platform*. The low level device driver communicates with the bus driver of an

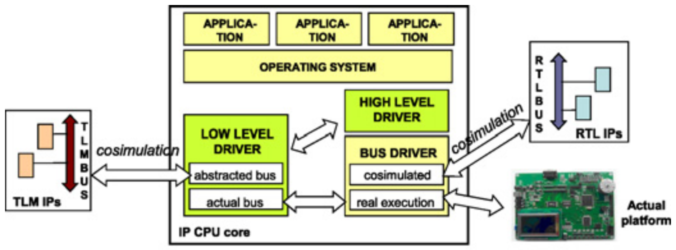


Fig. 13. Co-simulation flow versus real execution flow.

actual bus, but the platform is implemented at RTL level. The bus driver can be either the OS driver managing the bus or the load and store units, whenever the CPU is directly controlling the bus interface. Co-simulation is between the actual bus and the simulated platform. Thus, the low level driver behaves as if the RTL device was a synthesized device and the bus driver will hide co-simulation from the driver.

3) *Actual bus* and *real execution*. The target platform is synthesized and there is no need for co-simulation. The low level driver communicates with the device through a bus driver with I/O operations or with read and write operations on the device registers.

The choice has effect on the low level driver, that will handle the communication with the device with different approaches.

3) *Device Class:* By choosing a device class, such as Ethernet network devices, the designer selects a set of behaviors that the device driver must support, such as I/O queues or maintenance of statistics. This choice has effects on the functions implemented by the high level device driver. In this paper, we do not further focus on device class-specific feature specification, our main focus being on specific features concerning the target scenario. For device class categorization, state-of-art approaches can be applied, such as [13].

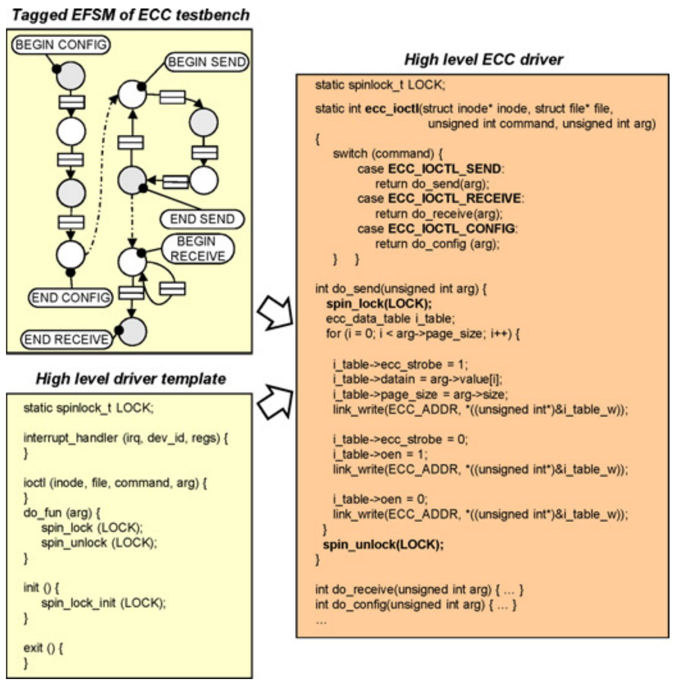
It is important to note that the methodology supports the generation of device drivers for simple character and network devices. Complex communication mechanisms (such as DMA and buffer management) and block devices have not been faced yet. However, this would imply to extend the device class support, whereas the core methodology would remain untouched.

D. Fourth Step: high level Driver Generation

The high level driver implements the communication protocol to correctly interact with the device and handles the interrupts identified by the user in the second step.

The previous steps have collected all the information needed. The device communication protocol has been obtained from the RTL testbench, the interrupts identified at the second step add interrupt handling resources and the architectural choices made in the third step might have effects on the handling of race conditions. The automatic generation of the high level driver consists of the following three main phases (Fig. 14).

1) *Data Table Generation:* A data table is generated in which all the RTL device ports are listed as well as the corresponding data size and type by parsing the testbench code. The data table is necessary to generate the driver data structures that provide support for exchanging data between the device driver and the device.



```
static spinlock_t LOCK;

interrupt_handler (irq, dev_id, regs) {
}

ioctl (inode, file, command, arg) {
}
do_fun (arg) {
    spin_lock (LOCK);
    spin_unlock (LOCK);
}

init () {
    spin_lock_init (LOCK);
}

exit () {
}
```

```
static spinlock_t LOCK;

static int ecc_ioctl(struct inode* inode, struct file* file,
                     unsigned int command, unsigned int arg)
{
   switch (command) {
        case ECC_IOCTL_SEND:
            return do_send(arg);
        case ECC_IOCTL_RECEIVE:
            return do_receive(arg);
        case ECC_IOCTL_CONFIG:
            return do_config (arg);
   }  }

int do_send(unsigned int arg) {
  spin_lock(LOCK);
  ecc_data_table i_table;
  for (i = 0; i < arg->page_size; i++) {

    i_table->ecc_strobe = 1;
    i_table->datain = arg->value[i];
    i_table->page_size = arg->size;
    link_write(ECC_ADDR, *((unsigned int*)&i_table_w));

    i_table->ecc_strobe = 0;
    i_table->oen = 1;
    link_write(ECC_ADDR, *((unsigned int*)&i_table_w));

    i_table->oen = 0;
    link_write(ECC_ADDR, *((unsigned int*)&i_table_w));
  }
  spin_unlock(LOCK);
}

int do_receive(unsigned int arg) { ... }
int do_config(unsigned int arg) { ... }
...
```

Fig. 14. Example of generation of the high level driver for the testbench reported in Fig. 5.

2) *Device Driver Functionality Generation:* Each of the EFSM subgraphs identified during the second step corresponds to a precise driver functionality (Section VI-B), used to fill in the high level template.

The subgraphs are automatically elaborated to obtain the high level driver functions by exploiting the data table structures previously generated. The functions of the generated device driver may be invoked by applications in an order that differs from the one followed in the starting testbench, thus increasing flexibility in the device driver usage.

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 the second step (Section VI-B), that field is set with the formal parameter values passed by the application. On the other hand, the methodology extracts data values for fields specific to the RTL interface, such as values for handshaking sequences, flags, burst cycles, etc. 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. Furthermore, the device driver does not constrain the order in which functionalities are invoked by the application.

The architectural choices made during the third step might add mutual exclusion resources, such as spinlock attainment and release.

Interrupt handling routine generation. If one or more interrupts have been identified in the testbench during the second step, the corresponding interrupt handling routines must be generated. As described in Section VI-A2, these routine will only wake up the driver (e.g., with a `wake_up_interruptible` primitive in any Linux OS).

Fig. 14 summarizes this methodology step. The subgraphs of the testbench EFSM representing a certain driver functionality (top of Fig. 14 on the left) are inserted into the driver template generated according the architectural configuration (bottom of Fig. 14 on the left). The result is the high level driver code (right of Fig. 14), implementing the device communication protocol.

E. Fifth Step: low level Driver Generation

The final step of the proposed methodology is the low level driver generation (Fig. 12). It relies on the template produced during the third step, which contains information about interrupt handling, mutual exclusion and target platform. The low level driver is common for all the high level drivers that rely on the same communication technology. Thus, at most one low level driver for each communication style must be generated.

The crucial choice for this step is the type of communication with the device. Real execution and co-simulation have great effects on the read and write primitives and on data handling.

If the target platform implies *co-simulation*, the low level driver will use the functions provided by the co-simulation environment to communicate with the simulated platform (Section III-D).

Else, with *real execution*, the RTL ports listed in the data table generated during the fourth step must be mapped to memory addresses, to implement a MMIO approach.

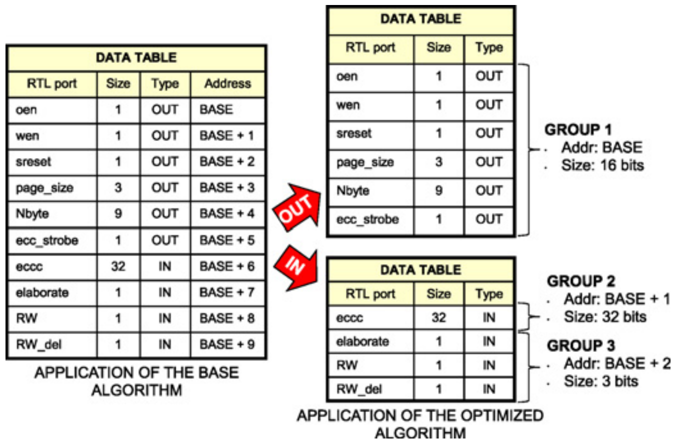
The RTL testbench accesses the RTL device functionality by writing to PIs and reading from POs. Once the device has been synthesized, the driver performs the corresponding tasks by writing to and reading from physical addresses. Thus, each port of the RTL device model is associated to a physical address assigned after synthesis and platform integration. In general, an address is associated to a device and the PIs/POs of the device are identified by adding an offset to such an address.

If the device is fixed or it has already been synthesized, then the methodology adopts the mapping to physical addresses embedded in the design. Else, if the device is configurable and it has not been synthesized yet, a mapping algorithm builds an optimized mapping to physical addresses, with the goal of saving memory and communication overheads.

The basic algorithm to assign physical addresses is the following. A device address is associated to each device port with port size no larger than a word. If the port exceeds the word size, it is associated to more than one address. The left side of Fig. 15 shows an example of application of this algorithm to the ECC device. This algorithm wastes bus cycles required to communicate with the device, since each write or read operation on a single port would require one entire bus cycle, even if only one significant bit is transferred.

To improve the quality of the device driver code, the methodology adopts an optimization of the assignment algorithm. Device ports are divided into two classes (input ports and output ports) and divided into groups such that the cumulative width is maximum 32 bits. A physical address is then assigned to each of these groups. This simple approach reduces the number of addresses used, as highlighted on the right side of Fig. 15.

Such an approach is recommended by HdS generation techniques to save address space and increase system performance [5] and it is feasible with the support of a particular C struct attribute, `__packed__`, used in structure declarations to assign the minimum memory required to each field. This would require also a redesign step of the device, to extract values from the `__packed__` structure and to map them to device ports. However, this straightforward step is balanced by benefits in terms of communication overhead.



RTL port	Size	Type	Address
oen	1	OUT	BASE
wen	1	OUT	BASE + 1
sreset	1	OUT	BASE + 2
page_size	3	OUT	BASE + 3
Nbyte	9	OUT	BASE + 4
ecc_strobe	1	OUT	BASE + 5
eccc	32	IN	BASE + 6
elaborate	1	IN	BASE + 7
RW	1	IN	BASE + 8
RW_del	1	IN	BASE + 9

RTL port	Size	Type
oen	1	OUT
wen	1	OUT
sreset	1	OUT
page_size	3	OUT
Nbyte	9	OUT
ecc_strobe	1	OUT

RTL port	Size	Type
eccc	32	IN
elaborate	1	IN
RW	1	IN
RW_del	1	IN

Fig. 15. Example of address table built for the ECC device and its optimization.

TABLE I
BENCHMARK CHARACTERISTICS

Device	PI (#)	PO (#)	Gates (#)	FF (#)	IP (loc)	Processes (#)	Modules (#)
ECC	25	32	993	79	174	3	1
CRC	56	34	9,213	385	589	18	1
DSPI	25	21	1,335	132	3,361	53	12
DIV	35	33	383	69	1,167	8	4
GCD	67	67	2,277	166	4,797	22	6
ROOT	36	33	319	110	870	6	4
FFT	92	114	87,397	1,359	3,335	27	1
ETH	93	42	46,113	4,955	2,307	9	1

VII. EXPERIMENTAL RESULTS

The goal of the experiments is to evaluate the automatic generation of device drivers in the context of the design space exploration of embedded platforms. After the explanation of the benchmarks, this section reports the results of the automatic driver generation in terms of driver characteristics. Then, the section concludes with the evaluation of the applicability and efficiency of the proposed methodology by generating device drivers for the design exploration of a case study platform.

A. Device Characteristics

The proposed methodology has been applied to generate the following set of device drivers for devices (i.e., IPs synthesized to FPGA) provided by STMicroelectronics:

1) an error correction code (ECC);
2) a cyclic-redundancy checking (CRC);
3) a synchronous peripheral interface (DSPI);
4) a filter for RGBA representation of pixels (DIV);
5) a device calculating the greatest common divisor (GCD);
6) a square root device (ROOT);
7) a Fast Fourier Transform (FFT);
8) a cs8900a Ethernet Controller (ETH).

The device testbenches, which have been provided by the device vendor, have not been modified to apply the proposed methodology, to preserve both the structure and the content of the code. This highlights the feasibility and the practicality of the methodology on industrial testbenches. Table I summarizes the characteristics of the devices and the corresponding RTL testbenches.

TABLE II
TESTBENCH CHARACTERISTICS

Device	Loc	EFSM States (#)	EFSM Trans. (#)	EFSM time (ms)	Wait (#)	Interrupts (#)
ECC	93	10	11	130	12	0
CRC	442	13	12	152	13	0
DSPI	462	49	52	843	43	1
DIV	71	3	4	56	3	0
GCD	47	6	8	87	2	1
ROOT	57	3	4	66	4	1
FFT	198	24	27	541	23	1
ETH	342	44	43	769	57	0

Columns *PIs* and *POs* show the number of primary inputs (PI) and primary outputs (PO) respectively. Column *Gates* reports the number of gates, whereas column *FF* shows the number of flip flops. The following columns outline the characteristics of each design, i.e., lines of code (column *IP (loc)*), number of processes (column *Processes (#)*) and number of modules (column *Modules*).

Table II shows the main characteristics of the testbenches used for each design and of the extracted EFSM. Column *Loc* shows the number of lines of SystemC code of the testbenches that have been analyzed by the parser for extracting the driver information. Columns (*EFSM States (#)*) and (*EFSM Trans. (#)*) presents the number of states and of transitions of the EFSM automatically extracted from the testbench itself. Column *EFSM time (ms)* shows the time needed to automatically extract the EFSM from the starting testbench. Column *Wait (#)* shows the number of wait statements used in the RTL testbench for sending data to the RTL device, which are then converted in a corresponding function call to the *write* operation implemented in the low level of the driver. Finally, column *Interrupts (#)* shows the number of signals selected as interrupt signals from the analysis of the self loop in the testbench code, as explained in Section VI-A2. This Table shows that automatic extraction of the EFSM of the testbench has been successfully obtained on all the designs and that the generation time was extremely low (less than 1 s for all designs).

B. Automatic Device Driver Generation

The proposed methodology has been implemented in D^2Gen, a tool built on top of HIFSuite [7]. D^2Gen currently supports the generation of device drivers targeting Linux operating systems. As future work, it will be extended to support other operating systems.

First, the starting VHDL and SystemC testbenches have been automatically translated to the HIF intermediate format by the HIFSuite front-end (*vhdl2hif* and *sc2hif*). Then, D^2Gen has been run on the HIF models. D^2Gen provides the user with a graphic interface to ease EFSM tagging and the specification of the target architecture configuration (see Section VI-C). The HIF description generated as a result by D^2Gen has been then converted to C code by using the HIFSuite *hif2c* back-end tool. The final C code is ready to be compiled and executed in the target (or co-simulated) architecture.

Table III reports the main characteristics of the generated device drivers. Column *CPU organ.* reports the CPU organizations presented in Section III for which the drivers have been generated. All drivers are generated for a Linux operating system.

TABLE III
CHARACTERISTICS OF THE GENERATED DRIVERS FOR EACH DIFFERENT CPU ORGANIZATION

Device	CPU organ.	D.Driver (loc)	D^2Gen time (s)
ECC	SP-SD	457	3.40
	SP-MD	457	3.42
	SMP-SD	469	3.43
	SMP-MD	481	3.45
	AMP-SD	457	3.41
	AMP-MD	457	3.42
CRC	SP-SD	421	3.49
	SP-MD	421	3.48
	SMP-SD	433	3.51
	SMP-MD	445	3.52
	AMP-SD	421	3.49
	AMP-MD	421	3.49
DSPI	SP-SD	609	4.34
	SP-MD	609	4.34
	SMP-SD	621	4.36
	SMP-MD	633	4.38
	AMP-SD	609	4.33
	AMP-MD	609	4.34
DIV	SP-SD	367	1.91
	SP-MD	367	1.92
	SMP-SD	379	1.93
	SMP-MD	391	1.95
	AMP-SD	367	1.92
	AMP-MD	367	1.90
GCD	SP-SD	389	1.72
	SP-MD	389	1.73
	SMP-SD	401	1.74
	SMP-MD	413	1.76
	AMP-SD	389	1.73
	AMP-MD	389	1.73
ROOT	SP-SD	398	1.39
	SP-MD	398	1.39
	SMP-SD	410	1.41
	SMP-MD	422	1.42
	AMP-SD	398	1.39
	AMP-MD	398	1.40
FFT	SP-SD	441	3.50
	SP-MD	441	3.51
	SMP-SD	453	3.52
	SMP-MD	465	3.54
	AMP-SD	441	3.53
	AMP-MD	441	3.53
ETH	SP-SD	432	4.05
	SP-MD	432	4.05
	SMP-SD	438	4.07
	SMP-MD	442	4.08
	AMP-SD	432	4.06
	AMP-MD	432	4.06

The total number of lines of code of each device driver is reported in column *D.Driver (loc)*. This highlights that D^2Gen successfully generated devices drivers for a variety of configurations and for devices of different characteristics and complexity.

Finally, column D^2Gen *time (s)* reports the time needed for the code generation process. This time includes only the automatic elaboration performed by D^2Gen to obtain the code. The time required both for the analysis of the testbench to identify the communication protocol and for determining the target architecture characteristics (such as, the CPU configuration) is not included. However, also the manual implementation of device drivers requires to study the device protocol specification and to determine the target architecture requirements. Thus, this analysis time is common to both the methodology proposed in this article and to traditional manual implementation flows.

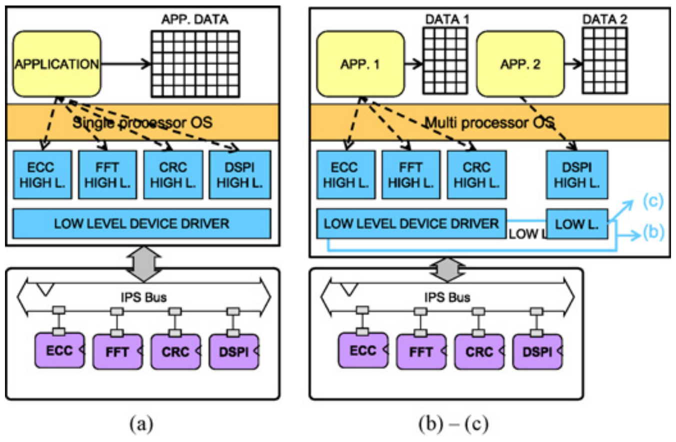


Fig. 16. Case study platform: (a) Single processor. (b) SMP multiple devices. (c) SMP single device combined with multiple devices.

The generation time is dominated by the EFSM extraction process. Table II shows that this step is almost instantaneous for the analyzed devices. The core claim is that more complex devices, with more complex testbenches, would lead to bigger EFSMs, with a larger number of states and transitions. However, the EFSM extraction algorithm scales well on complex testbenches, as highlighted in Section IV. Furthermore, even though the user may tag more and more complex functionalities, the underlying algorithm would still hold. A larger number of functions exported to the SW domain implies that a larger number of functionality subgraphs can be detected in the starting testbench. As a result, even though the number of subgraphs grows, the complexity does not explode, as the function generation steps and the subgraph elaboration are applied to single subgraphs (see Section VI-B). Thus, the entire semi-automatic generation methodology is promising to be applicable to more complex devices.

For each device, a few minutes of manual work have been spent for setting the parameters corresponding to the target CPU organization and for tagging the EFSM of the testbenches. Then, Table III shows that the driver generation by D^2Gen takes few seconds for each scenario. On the other hand, an average of 12 person-days have been spent for implementing the equivalent device drivers by hand.

It is important to note that the driver code generated by D^2Genmay not get the same performance as the code developed and optimized by programmers specialized in Hardware-dependent Software. However, this methodology goal is to provide a first implementation of the device driver, to allow quick integration in the target platform and, as a consequence, to support rapid design space exploration. Then, the device drivers may be further optimized and customized to the specific needs.

C. Semi-Automatic Driver Generation in Embedded Platform Exploration

This section aims at analyzing, with a case study, the impact of D^2Gen in automating the device driver generation in the context of embedded platform design exploration.

Fig. 16 depicts the analyzed case study. The SW application consists of data intensive elaborations, which rely on four IPs, i.e., ECC, FFT, DSPI and CRC. In the design exploration, different architecture configurations are evaluated: single processor in which the application is executed sequentially (Fig. 16(a)); a SMP multiple devices configuration where the application is decomposed on two tasks (Fig. 16(b)), and SMP single device combined with SMP multiple device configuration (Fig. 16(c)). Each configuration affects the platform performance in terms of synchronization overhead to access the devices and to manage computation.

TABLE IV
RESULT OF DESIGN SPACE EXPLORATION APPLIED TO THE EXAMPLE IN FIG. 16

	Single proc.	SMP-MD	SMP-SD
low level (loc)	160	160	312 (152+160)
high level (loc)	1472	1568	1556
Driver invoc. (#)	398	398	398
D^2Gen time (s)	14.75	14.89	14.85
Exec. time (s)	738	624	552

The different versions of device drivers have been instantaneously generated with $D^2Gen$by setting the corresponding scenario parameters. The results of such generation are outlined in Table IV. Column *Single proc.* shows data related to the single processor scenario, column *SMP-MD* refers to the SMP - multiple devices scenario, whereas column *SMP-SD* refers to the SMP - single device scenario. Lines *high level* and *low level level (loc)* report the overall lines of code of the high level and low level device drivers respectively. Line *Driver invoc. (#)* shows the number of invocations of the low level and high level drivers made by the application. Finally, line D^2Gen *gen. time (s)* shows the time required to generate all the device drivers, calculated as the sum of generation times for the four device drivers.

The application has been run with the three configurations by exploiting the cosimulation framework presented in [24] and outlined in Section III-D. In this way, it is possible to validate the generated device drivers and to estimate the application performance in the different platform scenarios. Line *Exec. time (s)* of Table IV reports the time necessary to run the application in the three scenarios.

Looking at the results, the designer can verify that the synchronization overhead on the SMP configuration decreases the performance gain of partitioning on multiple processors. Looking at the application, the designer may realize that one of the two parallel tasks communicates only with the DSPI device. When the low level driver is shared, the device driver requests from the second application are queued to the requests of the first one, even if they are accessing to different devices. Thus, the designer may want to generate the DSPI device driver with the single device configuration, to have a separate low level device driver devoted to the DSPI device (and thus to the second application). The result of this new generation step is shown in Table IV (Column *SMP-SD*). As highlighted by the execution time, the new solution reduces contention on the shared resources and thus it is preferred to the previous solutions.

This simple case study shows that D^2Gen supports fast configuration of device drivers and it offers further optimization directions in the design space exploration phase.

VIII. Concluding Remarks

In this article we presented an automatic device driver generation and customization methodology for rapid development of embedded platforms based on different CPU organizations. The methodology exploits the testbench provided with the

RTL model of the device to automatically generate the formal model of the device communication protocol. The device driver code is then customized to fulfill the requirements of the target platform, whose description is selected by the designer among a set of architectural alternatives. Synchronization primitives are added whenever needed by the specific CPU model and organization of the target platform. The generated drivers have been tested on a family of embedded platforms with different CPU organizations.

References

[1] STMicroelectronics. (2010). *SPEAr Embedded Microprocessors* [Online]. Available: http://www.st.com/stonline/products/families/embedded_mpu/embedded_mpus%.htm

[2] Freescale. (2009). *MPX4080: Integrated Pressure Sensor* [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPX4080

[3] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson, "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Proc. IEEE ISSCC*, 2010, pp. 19–21.

[4] J. Henkel, X. Hu, and S. Bhattacharyya, "Taking on the embedded system design challenge," *Computer*, vol. 36, no. 4, pp. 35–37, Apr. 2003.

[5] W. Ecker, W. Muller, and R. Domer., *Hardware-Dependent Software*. Berlin, Germany: Springer, 2009.

[6] T. Henzinger and J. Sifakis, "The discipline of embedded systems design," *Computer*, vol. 40, no. 10, pp. 32–40, Oct. 2007.

[7] EDALab s.r.l.. *HIFSuite* [Online]. Available: URL: http://www.hifsuite.com

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

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

[10] M. O'Nils and A. Jantsch, "Device driver and DMA controller synthesis from HW/SW communication protocol specifications," *ACM TODAES*, vol. 6, no. 2, pp. 177–205, 2001.

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

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

[13] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser, "Automatic device driver synthesis with Termite," in *Proc. ACM/IEEE SIGOPS*, Oct. 2009, pp. 73–86.

[14] T. Katayama, K. Saisho, and A. Fukuda, "Prototype of the device driver generation system for Unix-like operating systems," in *Proc. IEEE ISPSE*, Nov. 2001, pp. 302–310.

[15] Y.-T. Hsu, Y.-J. Wen, and S.-D. Wang, "Embedded Hardware/Software design and cosimulation using user mode Linux and SystemC," in *Proc. IEEE ICPPW*, Sep. 2007, pp. 17–22.

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

[17] F. Merillon, L. Reveillere, C. Consel, R. Marlety, and G. Muller, "Devil: An IDL for hardware programming," in *Proc. OSDI*, vol. 4. Oct. 2000, p. 2.

[18] G. Schirner, A. Gerstlauer, and R. Domer., "Automatic generation of HdS for MPSoCs from abstract system specifications," in *Proc. ASP-DAC*, 2008, pp. 271–276.

[19] V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with RevNIC," in *Proc. ACM SIGOPS/EuroSys*, 2010, pp. 167–180.

[20] N. Bombieri, F. Fummi, G. Pravadelli, and S. Vinco, "Correct-by-construction generation of device drivers based on RTL testbenches," in *Proc. ACM/IEEE DATE*, 2009, pp. 1500–1505.

[21] A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer*, vol. 38, no. 2, pp. 63–69, Feb. 2005.

[22] G. K.-H. J. Corbet and A. Rubini, *Linux Device Drivers*. O'Reilly, 2009.

[23] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Norwell, MA, USA: Kluwer Academic, 2003.

[24] F. Fummi, G. Perbellini, D. Quaglia, S. Saggin, and S. Vinco, "Mixing simulated and actual hardware devices to validate device drivers in a complex embedded platform," in *Proc. IEEE MTV*, Dec. 2009, pp. 63–68.

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

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

[27] D. Bresolin, G. Di Guglielmo, F. Fummi, G. Pravadelli, and T. Villa, "The impact of EFSM composition on functional ATPG," in *Proc. IEEE DDECS*, Apr. 2009, pp. 44–49.

[28] Transaction Level Modeling Working Group. *OSCI TLM 2.0* [Online]. Available: http://www.systemc.org

Andrea Acquaviva (M'03) received the Ph.D. degree in electrical engineering from the University of Bologna, Bologna, Italy, in 2003.

From 2001 to 2003, he was a Research Intern with Hewlett Packard Labs, Palo Alto, CA, USA. From 2005 to 2007, he was a Visiting Researcher with the Ecole Polytechnique Federale de Lausanne, Lausanne, Vaud, Switzerland. Since 2008, he has been with the Department of Computer Engineering and Automation, Politecnico di Torino, Turin, Italy. His research (between 2000 and 2012) yielded more than 100 papers in international journals and peer-reviewed international conference proceedings. His current research interests include parallel computing, sensor networks, and computational biology.

Nicola Bombieri (M'05) received the Ph.D. degree in computer science from the University of Verona, Verona, Italy, in 2008.

Since 2008, he has been a Researcher and Assistant Professor with the Department of Computer Science, University of Verona. He has been involved in several national and international research projects and has published more than 50 papers on conference proceedings and journals. His current research interests include TLM design and verification of embedded systems and automatic generation and optimization of embedded SW.

Franco Fummi (M'92) received the Ph.D. degree in electronic engineering from the Politecnico di Milano, Milan, Italy, in 1995.

He has been the Head of the Department of Computer Science, University of Verona, Verona, Italy, since 2012. He was a Full Professor at the Department of Computer Science, University of Verona. Since 1995, he has been with the Department of Electronics and Information, Politecnico di Milano, as an Assistant Professor. In July 1998, he became an Associate Professor in computer architecture at the Department of Computer Science, University of Verona. His current research interests include electronic design automation methodologies for modeling, verification, testing, and optimization of embedded systems.

Sara Vinco (M'09) received the Masters degree in computer science from the University of Verona, Verona, Italy, in 2009. She is currently pursuing the Ph.D. degree in computer science at the University of Verona.

During the Ph.D. degree, she spent six months as a visiting student at the University of Michigan, working with Prof. V. Bertacco from June 2011 to December 2011. She is currently a Post-Doctoral Research Associate at the Department of Computer Science, University of Verona. Her current research interests include electronic design automation methodologies for simulation and validation of heterogeneous embedded systems.