Using Virtual Prototypes to Improve the Traceability of Critical Embedded Systems

Jean-Michel Fernandez
Embedded Systems
Magillem Design Services
Paris, France
fernandez@magillem.com

Larry Lapides
Embedded Software
Imperas Software
Oxfordshire, United Kingdom
larryl@imperas.com

Abstract — This paper explains how the combination of innovative traceability techniques with advanced Virtual Prototyping execution environment helps detecting and locating critical embedded system bugs located at the frontier of Hardware and Software, by tracing the dependencies between all the objects of any kind such as requirements, specification, documentation, hardware or software meta-data.

Keywords — Embedded System; Traceability; Requirement; Specification; Design; Documentation; Virtual Prototype; Software; Hardware; SystemC; IP-XACT

I. Introduction

Designing critical systems requires compliance with domain specific safety standards such as DO-178B/C for avionics or ISO-26262 for automotive, which in turn require strong traceability from the functional specification down to the implementation of the complete system. Traceability is one of the essential activities of requirements management: it is used to ensure that the right product is being built at each phase of the embedded systems development life cycle, to measure the progress of that development and to reduce the effort required to determine the impacts of requested changes.

Efficient tools exist to trace requirements by creating and managing specification-implementation links. But to unambiguously ensure the completeness of a requirement, these tools usually miss the links between the hardware objects (such as an interrupt signal or a register), the software objects (such as a safe routine) and their execution state. Moreover, in order to record the execution state of an object, this object must be observable.

Adding a link to the design execution on the physical prototyping board may come too late in the design cycle to capture functional specification or requirements errors. Moreover, some of the objects (register value, interrupt signal) may not be observable on the physical prototype, or corner case states may not be easily achievable, leading to the inability to create or observe an error.

This paper describes how Virtual Prototypes (VPs) can be used to create the missing link between the functional requirements and their validation. First it describes how VPs can help verifying functional requirements. Then it details how traceability techniques can be combined with VPs to improve the debug of bugs that sit at the frontier between the Hardware (HW) and the embedded Software (SW). Last it shows on a typical use case how these combined techniques could help to quickly locate a system bug and discusses improvement areas of this work.

II. Virtual Prototypes

A. *Verifying functional requirements*

Functional requirements come from a usually informal analysis process to turn raw, incomplete requirements as elicited from the system stakeholders into a structured system requirements specification document. Various techniques exist to verify functional requirements ranging from manual reviews and inspection to formal verification and validation (V&V), using or not prototype or test designs.

The Validation process ensures that the system being developed or changed will satisfy its stakeholders, and that the system requirements specifications meet the stakeholders' goals and requirements. The Verification process ensures that each step followed in the process of building the embedded system (software and hardware) yields to the right product and that the requirement specifications are consistent with the refined implementation: functional model, accurate design, physical implementation.

Prototyping is a common process to help stakeholders (end users and customers) discover problems by validating and verifying their requirements: it is more accessible than the system specification, it demonstrates the requirements, it is reusable and evolutive. It can take various forms ranging from a paper prototype of a computerized system to a formal executable model of the specifications.

B. Using Virtual Prototypes

VPs are fully functional software simulation models of complete hardware systems that can execute unmodified production binary code at near real time speed. VPs therefore enable early functional validation of embedded software on the target hardware platform, usually months before the physical prototype is available. In addition, because of the nature of simulation, VPs offer controllability, observability and flexibility. VPs are usually based on SystemC standard [1]. In addition to the simulation environment, VP tools provide the necessary debug, monitoring or analysis features; usually implemented in a non-intrusive manner, without modifying or instrumenting the production code.

III. Integrating Specification, Design and Documentation

Magillem [3] tools use IP-XACT [2] as a pivot metadata to represent both HW and embedded SW. IP-XACT is a standard with a strong semantic that can be seen as a documentation format that references data from potentially multiple heterogeneous views. It was originally designed to represent Hardware components, but could equally be used to represent any object that communicates through interfaces. In addition, Magillem tools use a proprietary metadata to encompass all the standards and de-facto standard document formats based on XML such as DITA or Microsoft Office formats. XML-based formats (such as IP-XACT, DITA or Docx) allow the processing of its content by a tool. This mechanism allows Magillem to represent any document fragments and manage any link between objects of any kind.

A. Creating the VP

Imperas [4] and OVP [5] provide all the SystemC building blocks to quickly build a Virtual Prototype of an embedded system. These blocks can be automatically packaged into IP-XACT metadata with Magillem tool and the hardware system can be seamlessly assembled, compiled and simulated together with the embedded Software in a unified Eclipse framework.

B. Creating the links

Magillem provides an intuitive framework for creating all the links between any fragments of documentation (requirements, specification, code, datasheet…). The fragment can be as detailed as needed, ranging from a complete document (e.g. the specification of the hardware LED controller) to the finest unit object (e.g. the LED voltage or color parameter). Once the association is done the tool is able to analyze the impact of any change in any of the linked objects.

C. Debugging the Software

One of the most typical usages of a Virtual Prototype is to help writing, debugging and analyzing the embedded Software. Software developers spend a large part of their time debugging the SW. Usually the bug comes at the boundary between Hardware and Software, and it is only visible at run time. VP is a perfect tool for finding such bugs because it provides a very good observability of the HW registers and signals (e.g. is this interrupt signal raised when writing to that register?). Such debugging capabilities are usually provided by traditional VP tools and help capturing many SW errors.

D. Locating the errors

But sometimes, the bug is not there: the value of the register is correct and the interrupt signal is properly raised as described in the model; the problem may come from a misinterpretation of a requirement that led to an incorrect specification and an incorrect implementation of the behavior. In this case, the debugging tool is not enough; it has to be coupled with a traceability tool capable of tracing the path from the requirement, through the specification down to the register implementation. Such a trace would for example show that the register could only be written during the boot mode.

Linking the VP to the IP-XACT representation helps accessing data that is not available in the VP such as datasheet or non-functional properties such as power, voltage or frequency. Linking the IP-XACT representation to the requirements and specification documents allows capturing such misinterpretation errors that would have taken hours or days to locate otherwise. Sometimes the error comes from a change (in the model or in the requirement) that was not properly propagated throughout the traceability chain.

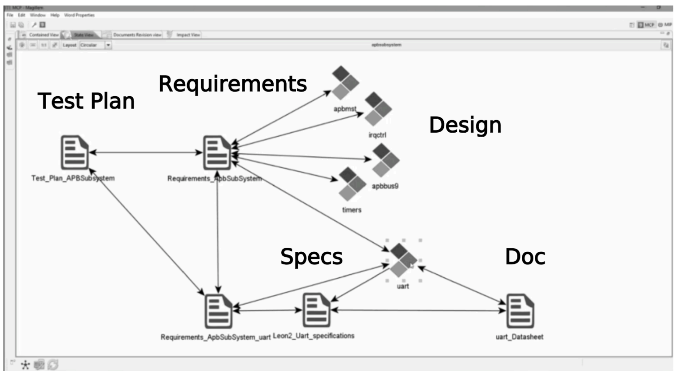

Fig. 1. Example of links between Requirements, Specification, Design, Documentation and test plan

IV. Case study: the LED system

Such advanced methodologies that mix SW debug on VP together with Traceability of requirements down to the HW and SW implementation has been validated on a simple system that controls the execution of critical tasks.

We have used a system based on the tutorial defined in [7]. It is composed of a micro controller based on an ARM M3 processor, including simple memories, a simple interconnect, a bank of 8 LEDS and a UART connected to a

Terminal display. The system can run a RTOS and an application that monitors the switching tasks. Each executing task is represented by a LED; the LED is ON (highlighted) when the task is active (i.e. running) and OFF when it is suspended or stopped. An extra (red) LED is highlighted when an error occurs.

A. *Requirements*

The LED Functional Requirements includes the lines defined in *Table I*.

TABLE I. LED REQUIREMENTS (SAMPLE)

Req number	Description
R-L1.1	The LED shall be used to indicate the system status.
R-L1.1.1	A flashing green or yellow LED shall indicate that the system is running as expected
R-L1.1.2	A flashing red LED shall indicate a fault condition.
R-L1.1.3	The correct LED shall flash on and off once every second. This flash rate shall be maintained to within 50ms.

B. *Refined requirements*

The system shall be based on the FreeRTOS [6] that includes the concept of co-routines and tasks. Tasks will be used for the Terminal display and co-routines for the LED display. The application code running on the RTOS shall create five flash co-routines and three tasks. One extra task (the idle task) is responsible for launching all the co-routines. The *Table II* details the mapping between the tasks and the LEDs.

TABLE II. LED REFINED REQUIREMENTS (SAMPLE)

Req number	Description
RR-L1.1	The flash co-routines control LED's zero to four.
RR-L1.2	LED five is toggled each time the string is transmitted on the UART.
RR-L1.3	LED six is toggled each time the string is correctly received on the UART.
RR-L1.4	LED seven is latched on when an error is detected in any task or co-routine. The error is detected by a check function (called by the idle task) that loads the general purpose registers with a known value, then checks each register to ensure the held value is still correct. As a low priority task this checking routine is likely to get repeatedly swapped in and out. A register being found to contain an incorrect value is therefore indicative of an error in the task switching mechanism.

These requirements are then further refined into low level HW and SW specifications.

C. *Refined specifications*

1) *HW specifications*

The UART and the LED peripherals have been derived from those defined in the TI Stellaris platform [8]. When the SW writes to the UART Data Register (DR), an interrupt is raised that will launch a SW interrupt routine. The LEDs are implemented as an 8 bits register. Each bit represents a LED: a one means highlight is ON, a zero means it is OFF. The flash co-routines are mapped to the bits 0 to 4 and are represented by a green LED. The UART tasks are mapped to the bits 5 and 6, represented by a yellow LED. And the error task is mapped to the bit 7 and represented as a red LED.

2) *SW specifications*

To control the HW peripherals, SW drivers have to be implemented. A sample of the LED driver is given in *Fig 2*.

```
void LedInitialise( void ) {...}
void LedSet( unsigned int LED, boolean value )
{
 unsigned char ucBit = ( unsigned char ) 1;
 vTaskSuspendAll();
 { /* atomic section */
   ucBit = ( ( unsigned char ) 1 ) >> LED;
   if( ! value ) {
     ucBit ^= ( unsigned char ) 0xff;
     ucOutputValue &= ucBit;
   } else {
     ucOutputValue |= ucBit;
   }
   ledWrite(ucOutputValue);
 } /* end atomic section */
 xTaskResumeAll();
}
```

Fig. 2. Sample of the SW driver code for the LED

These drivers access the HW registers through some Hardware Abstraction Layer (HAL) code, usually part of the Hardware BSP, as illustrated in *Fig 3*.

```
#define LED_BASE_ADDRESS 0x40004000
#define LED() *((volatile char *) LED_BASE_ADDRESS+4)
void ledWrite(unsigned char value) {
   LED() = value;
}
```

Fig. 3. Sample of the HAL code for the LED

The application SW specifications defining how the tasks and co-routines are created are not described here; the focus of this paper being the HW dependent SW.

D. *VP implementation*

A VP platform has been created using Imperas/OVP SystemC models for each IP defined in the specification: an ARM M3 instruction accurate fast processor model, a LED controller connected to a LED display and a UART connected to a Terminal. The Processor and the peripherals are connected to a simple interconnect and communicate through transactional interfaces. The UART interrupt signal is directly connected to the ARM core Interrupt Controller.

Each SystemC IP model has been automatically packaged in IP-XACT XML format to ease its management and reuse over time. These IP-XACT IP blocks have then been assembled and executed using Magillem VP assembly tool, as illustrated in *fig 4*.

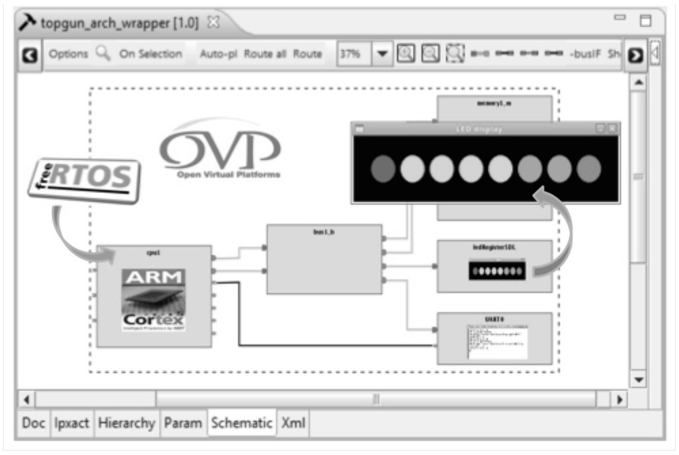

Fig. 4. Virtual Prototype of the LED platform

E. *Links creation*

Links have to be created between the VP (the design part) and the Requirements. Note that the specification documents have been omitted here to simplify the system. The creation of the links can be achieved with the Magillem tools by directly importing the Requirements and the IP-XACT representation of the VP. Links between fragments of the requirement document and the IP can then be easily created by simple drag and drop of fragments of data and visualized with the tool.

For example, the LED requirement R-L1.1.2 can be linked to the refined requirement RR-L1.4 itself linked to last bit field of the LED HW register and it can also be linked to the error routine on the SW side that updates the LED register.

F. *Debugging the system*

Now comes the exciting part of the work. When simulating this simple system, we observe that the red LED is highlighted after some time. The VP flexibility allows to simply putting a breakpoint in both the SW and in the HW when the LED register is written and stop the simulation when the red LED is highlighted.

Thanks to the link to the requirement RR-L1.4, we can see that the LED seven is mapped to the Error condition. But the register value shows that the bit0 is at 1 and the bit7 is at 0. It is very likely that either the display LEDs connected to the register have been reversed or that on the SW side, the mapping between the register bits and the tasks/co-routines was reversed. More debugging demonstrated that the SW side was the root of the error. The location of the SW error was in the LED driver: the LedSet function was erroneously shifting right instead of shifting left, as illustrated in *Fig 5*.

```
v void LedSet( unsigned int LED, boolean value )
{
  …
    ucBit = ( ( unsigned char ) 1 ) << LED; // error was here
  …
}
```

Fig. 5. Sample of the SW driver code for the LED

Both the SW and the HW were right. The error only shows up when linking the two and execute the SW with the HW. A direct pointer to the requirement could immediately separate out the HW and the SW responsibilities, saving hours of debug and iterations between HW and SW teams.

FUTURE WORK

Such an integrated environment with immediate impact analysis to help debugging complex systems is even more useful when some requirement changes or when the spec changes or when the implementation changes (e.g. when a bug is fixed). Of course this assumes that the links have been properly created and that they fully cover the requirements. Additional techniques need to be developed to automate the creation of the links and to verify the links are complete.

CONCLUSION

This paper explained how the combination of innovative traceability techniques with advanced VP execution environment helps locating SW errors by tracing the dependencies all the way through from requirements down to the embedded system execution and vice-versa. This is the beginning of a long avenue of developments to improve the consistency and coherency between the functional requirements and their implementation through early validation on VPs.

REFERENCES

[1] SystemC IEEE 1666-2011 specification, http://ieeexplore.ieee.org/document/6134619/

[2] IP-XACT IEEE 1685-2009 specification, http://ieeexplore.ieee.org/document/5417309/

[3] Magillem, www.magillem.com

[4] Imperas, http://www.imperas.com

[5] Open Virtual Platform (OVP), http://www.ovpworld.org/

[6] FreeRTOS, FreeRTOS.org

[7] Real Time Application Design Tutorial, http://www.freertos.org/tutorial/

[8] Texas Instrument Stellaris LM3S microcontrollers, http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit