Using Virtual Prototypes to Improve the Traceability of Critical Embedded Systems

Jean-Michel Fernandez, Magillem
Larry Lapides, Imperas
Agenda

- Scope
- Traceability
- Virtual Prototyping
- Linking the two worlds to improve the debug of Hardware-dependent-Software
- Illustration on a simple use case
- Future work and conclusion
Designing critical embedded 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 including Hardware and Software.
Traceability is one of the essential activities of requirements management:

- Ensures that the right product is being built at each phase of the embedded systems development life cycle,
- Measures the progress of that development
- Reduces the effort required to determine the impacts of requested changes.
Traceability tools miss some links

Efficient tools exist to trace requirements by creating and managing specification-to-implementation links.

But to unambiguously ensure the completeness of a requirement, these tools usually miss the links between the requirement specification, the hardware objects, the software objects and their execution state.
Traceability tools miss some links

- **Requirements**: R1: Launch safe routine when Secure register is accessed

- **HW objects**: Register, Interrupt signal...
  - Register Written?
  - Interrupt Raised?
  - Safe routine Launched?

- **SW objects**: Driver, Safe routine...
  - Executions state
This clearly illustrates the need for Integrating Specification, Design and Documentation (ISDD), a novel approach invented by Magillem.
Such an integration is enabled by the use of XML Metadata to represent any document fragments and manage any link between objects of any kind.

- **IP-XACT** (IEEE 1685) is an XML standard with a strong semantic to represent Hardware components potentially composed of multiple heterogeneous data
- **META-X©** is an XML innovative format that encompasses all the standards and de-facto standard document formats based on XML such as DITA or Microsoft Office formats.
Linking to HW details

Adding a link to the HW design execution on the physical prototyping board may come too late in the design cycle to capture functional specification or requirements errors.

Difficult to verify some functional requirements (e.g. inject/observe an error)

- Some objects may not be observable on the physical prototype (e.g. register value, interrupt signal)
- Some corner case states may not be reachable
Moreover, in order to record the execution state of an object, this object must be observable.
Prototyping

Prototyping is a common process to help stakeholders 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.
Virtual Prototyping

VPs are fully functional software simulation models of complete hardware systems that can execute unmodified production binary code at near real time speed.

VPs enable early functional verification & debug of embedded software on the target hardware platform, usually months before the physical prototype is available.
Virtual Prototyping

- VPs can be seen as an executable specification.
- They are based on a fast simulator and on fast simulatable models of the hardware system, usually using the SystemC standard (IEEE 1666).
- VPs offer controllability, observability and flexibility.
- VP tools provide the necessary debug, monitoring and analysis features; usually implemented in a non-intrusive manner, without modifying or instrumenting the production embedded SW code.
Creating the VP

- Imperas and OVP 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.
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 an hardware controller device) to the finest unit object (e.g. the voltage or throughput parameter).

Once the association is done the tool is able to analyze the impact of any change in any of the linked objects.
Creating the Links

- Requirement change
- Design change
- Impact analysis
- Documentation consolidation
- Impact consolidation
One of the most typical usages of a Virtual Prototype is to help writing, debugging and analyzing the embedded SW. Usually the bug comes at the boundary between HW and SW, 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.
- Such debugging capabilities are usually provided by traditional VP tools and help capturing many SW errors.

But sometimes, the bug is not there...
Locating the errors

The error may come

- From a change (in the model or in the requirement) that was not properly propagated throughout the traceability chain.
- Or from a misinterpretation of a requirement that led to an incorrect implementation of the behavior.

The debugging tool must therefore be coupled with a traceability tool capable of tracing the path from the requirement, through the specification down to the HW implementation.

- Such a trace would for example show that the register could only be written during the boot mode.
Magillem provides such an environment

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.

Locating the errors
Case study: overview

Simple system that controls the execution of critical tasks

System description

• Each executing task is represented by a LED
• The LED is ON (highlighted) when the task is active (i.e. running)
• The LED is OFF when it is suspended or stopped.
• An extra (red) LED is highlighted when an error occurs.
# Case study: System Requirements

<table>
<thead>
<tr>
<th>Req number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR-1.0</td>
<td>The system shall be based on the FreeRTOS that includes the concept of co-routines and tasks.</td>
</tr>
<tr>
<td>SR-1.1</td>
<td>Tasks will be used for the Terminal display and co-routines for the LED display.</td>
</tr>
<tr>
<td>SR-1.2</td>
<td>The application code running on the RTOS shall create five flash co-routines and three tasks.</td>
</tr>
<tr>
<td>SR-1.3</td>
<td>One extra task (the idle task) is responsible for launching all the co-routines.</td>
</tr>
<tr>
<td>SR-1.4</td>
<td>The system will run on a Micro-controller based on an ARM M3 processor, connected to a bank of 8 LEDs and to a UART controlling the Terminal display.</td>
</tr>
<tr>
<td>R-L1.1</td>
<td>The LED shall be used to indicate the system status.</td>
</tr>
<tr>
<td>R-L1.1.1</td>
<td>A flashing green or yellow LED shall indicate that the system is running as expected.</td>
</tr>
<tr>
<td>R-L1.1.2</td>
<td>A flashing red LED shall indicate a fault condition.</td>
</tr>
<tr>
<td>R-L1.1.3</td>
<td>The correct LED shall flash on and off once every second. This flash rate shall be maintained to within 50ms.</td>
</tr>
</tbody>
</table>
## Case study: Refined Requirements

### LED Functional Requirements (sample)

<table>
<thead>
<tr>
<th>Req number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR-L1.1</td>
<td>The flash co-routines control LED's zero to four.</td>
</tr>
<tr>
<td>RR-L1.2</td>
<td>LED five is toggled each time the string is transmitted on the UART.</td>
</tr>
<tr>
<td>RR-L1.3</td>
<td>LED six is toggled each time the string is correctly received on the UART.</td>
</tr>
<tr>
<td>RR-L1.4</td>
<td>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.</td>
</tr>
</tbody>
</table>

### LED Requirements (sample)

<table>
<thead>
<tr>
<th>Req number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-L1.1</td>
<td>The LED shall be used to indicate the system status.</td>
</tr>
<tr>
<td>R-L1.1.1</td>
<td>A flashing green or yellow LED shall indicate that the system is running as expected.</td>
</tr>
<tr>
<td>R-L1.1.2</td>
<td>A flashing red LED shall indicate a fault condition.</td>
</tr>
<tr>
<td>R-L1.1.3</td>
<td>The correct LED shall flash on and off once every second. This flash rate shall be maintained to within 50ms.</td>
</tr>
</tbody>
</table>
Case study: HW Specifications

<table>
<thead>
<tr>
<th>Req number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR-S-2.1</td>
<td>When the SW writes to the UART Data Register (DR), an interrupt is raised that will launch a SW interrupt routine.</td>
</tr>
<tr>
<td>RR-L-2.1</td>
<td>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.</td>
</tr>
<tr>
<td>RR-L-2.2</td>
<td>The UART tasks are mapped to the bits 5 and 6, represented by a yellow LED.</td>
</tr>
<tr>
<td>RR-L-2.3</td>
<td>The error task is mapped to the bit 7 and represented as a red LED.</td>
</tr>
<tr>
<td>RR-L-2.4</td>
<td>The flash co-routines are mapped to the bits 0 to 4 and are represented by a green LED.</td>
</tr>
</tbody>
</table>

LED Hardware Requirements (sample)
## Case study: SW specifications

<table>
<thead>
<tr>
<th>Hardware Abstraction Layer (HAL) code, usually part of the Hardware BSP</th>
</tr>
</thead>
</table>
| 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;
|     }
|   } /* end atomic section */
|   ledWrite(ucOutputValue);
|   xTaskResumeAll();
| }
| #define LED_BASE_ADDRESS 0x40004000
| #define LED() *((volatile char *) LED_BASE_ADDRESS + 4)
| void ledWrite(unsigned char value) {
|   LED() = value;
| }
| 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. |

**SW Driver for the LED peripheral**

```c
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;
        }
    } /* end atomic section */
    ledWrite(ucOutputValue);
    xTaskResumeAll();
}
#define LED_BASE_ADDRESS 0x40004000
#define LED() *((volatile char *) LED_BASE_ADDRESS + 4)
void ledWrite(unsigned char value) {
    LED() = value;
}
```
**Case study: VP implementation**

- VP platform based Imperas/OVP SystemC TLM models for each IP defined in the specification
- Each SystemC IP model has been automatically packaged in IP-XACT XML format
Case study: Links creation

Links have been created between the VP (the HW/SW design part) and the Requirements.
Case study: System debug

When simulating, 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.
Case study: System debug

Thanks to the link to the requirements, we can see that the LED seven is mapped to the Error condition.

And that in normal mode (no error) the value of the LED7 (bit7 of the LED register) should 0 (not 1).
Case study: Conclusion

More debugging demonstrated that the SW side was the root of the error.

The location of the SW error was in the LED driver: shift right instead of shift left.

`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;
    }
    ...
    ...

The error task is mapped to the bit 7 and represented as a red LED.
Case study: Conclusion

The error only shows up when linking together the Specification and the Implementation, and when executing 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.
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.
Thank you

More on magillem

stand 4-538