Modeling of Hardware and Software for specifying Hardware Abstraction Layers

Yves Bernard, Cédric Gava, Cédrik Besseyre, Bertrand Crouzet, Laurent Marliere, Pierre Moreau, Samuel Rochet

Modeling of Hardware and Software for specifying Hardware Abstraction Layers

Yves BERNARD[1], Cédric GAVA[2],
Cédrik BESSEYRE[1], Bertrand CROUZET[1], Laurent MARLIERE[1], Pierre MOREAU[1], Samuel ROCHET[2]

(1) Airbus Operations SAS
(2) Subcontractor for Airbus Operations SAS

Abstract

In this paper we describe a practical approach for modeling low level interfaces between software and hardware parts based on SysML operations. This method is intended to be applied for the development of drivers involved on what is classically called the "hardware abstraction layer" or the "basic software" which provide high level services for resources management on the top of a bare hardware platform. It is also an enabler for co-design processes since the design of hardware and software can be decoupled. In addition this approach is compatible with virtual prototyping technologies such as SystemC/TLM. An application to a simple a study case is provided for illustration purpose.

Keywords: MBSE, System Engineering, SysML, model, Interface, hardware/software co-design

1. INTRODUCTION

For many software developers, an application is based on the services provided by an operating system and/or a middleware that fully abstracts the actual implementation of the hardware platform on which this application runs. However there is always a point, where the most basic software, the one that directly access to the hardware resource, has to be developed. Usually, the design of the high level software applications uses distinct domains and different abstractions compared with the design of hardware platforms. This cannot be the case when designing this basic software. The lowest level interface between the hardware components and the software has to be built on common concepts. There is neither middleware, nor operating system in this context. Resources are provided by bare hardware components, only described in their datasheets. Among them, there are the processors with theirs sets of instructions used by software for specifying execution flows.

Often, the industrial context on which the development is taking place is time-constrained in such a way that the design of the basic software shall generally be carried out concurrently with the design of the hardware platform. It is the so-called: "hardware/software co-design" issue.

A common practice is to organize the various components of a product according to layers of responsibilities. It enables the separation of concerns and facilitates the re-use, the maintenance and the work assignment. Concepts like: Hardware Dependent Software (HDS) and Hardware Abstraction Layer (HAL) are widely used for the development of such systems [1].

The idea of a unified representation for both the hardware and the software, allowing a seamless approach for their development was presented in [2]. In [3], [4], [5] or [6] such models are developed for the purpose of trade-offs analysis regarding the allocation of computing tasks in the context of Systems on Chip (SoC). The hardware(HW)/software(SW) interface is abstracted in SystemC in such a way that it is easy to change the allocation of a functional task between hardware and software resources, but the modeling of HDS is not actually broached.

In [3], [4] and [7], SystemC models are used in some later steps of the design process to validate the HDS, but only the HW is modeled. Indeed in these models, an Instruction Set Simulator (ISS) can execute the software, directly in its binary compiled format, almost as a real platform does. In these approaches, nothing is said about the modeling of the HDS: the topic is to debug the software as a whole.

Finally, all the methods involving the UML [8] and MARTE [9] standards, like [10], [11], [12] or [13], use allocation relations to link software tasks to resources of the hardware. From the point-of-view of software, these relations enable trade-off analysis, typically: tasks scheduling or competition for shared resources. These methods assume that applications will rely on an Operating System and HDS for simplifying the access to hardware resources by abstracting the actual hardware topology and protocols. However they do not deal with the design of those essential software components.

The kind of the hardware models involved in hardware/software co-design is focused on the digital functions of an electronic circuit related by interconnection components like buses or bridges. The modeling of such architectures, which we will call "digital architecture" in this paper, has been addressed by standards such as IP-XACT [14] and SystemC [15]. The IP-XACT standard is firstly designed for assembling IPs. It provides a structural description of those architectures but it does not address the behavioral aspects. The SystemC standard provides facilities for describing both the structure and the behavior of hardware architectures. Derived from the C++ programming language, SystemC is executable and, associated with an Instruction Set Simulator (ISS), it makes it possible to runs co-simulations of software on virtual hardware platforms. However, since they are focused on execution, SystemC models are not convenient to support all the aspects of a design model. It represents a possible implementation of a specification that provides the expected execution traces as long as real-time aspects remains out-of scope. In addition, the level of details required by SystemC model makes them inappropriate for the early stages of the HDS design phase. SysML [16] is better adapted for this purpose; this is the reason why we chose it for implementing the modeling methodology presented in the following paragraphs.

2. VIEWPOINTS

2.1.1 The hardware concepts

Each function of a digital architecture is modeled as a reusable component that can owns ports which specify connection points between the component and its environment. There are two fundamental roles of ports in these architectures: "master" ports, which can initiate a communication, and "slave" ports, which can only react to communications requests. For each port an "interface definition" is provided. It usually refers to a communication standard like PCI, Ethernet or AFDX. If the communication conveys identification (i.e. address) information along with data, then the ports conforming to this interface definition are said to be "addressable".

For each addressable slave port, a memory map can be established. It specifies how any resource can be unambiguously identified and thus read or written at that port through an addressable master port.

Digital architectures assemble two kinds of components:

- Functional components like: data storages, processors or input/output devices which provide the resources used by the software applications
- Interconnection components, like "bridges" and "buses" which provide the communication channel allowing all the functional components working together as a whole (i.e. the so-called "platform"). In general, the word "bus" is used for interconnection components which have ports of the same interface definition, whereas "bridges" is used for those having ports of different interface definitions[1].

"Transparent" interconnections can map storage elements in slave's memory maps into masters address spaces whereas "opaque" do not .When more than one resource is published at a slave port, their precise locations are characterized by offset values which are used in address resolution. We call "address resolution" of an element E_S owned by the memory map MM_S of a slave port S, the computation of an address integer value $A_{ES\text{-}M}$ in a master's address space AS_M of a master port M. This computation mainly depends on the offset $O_{ES\text{-}S}$ of E_S in the memory map MM_S and on the offsets $O_{Mi1}$of the traversed master port between M and S.

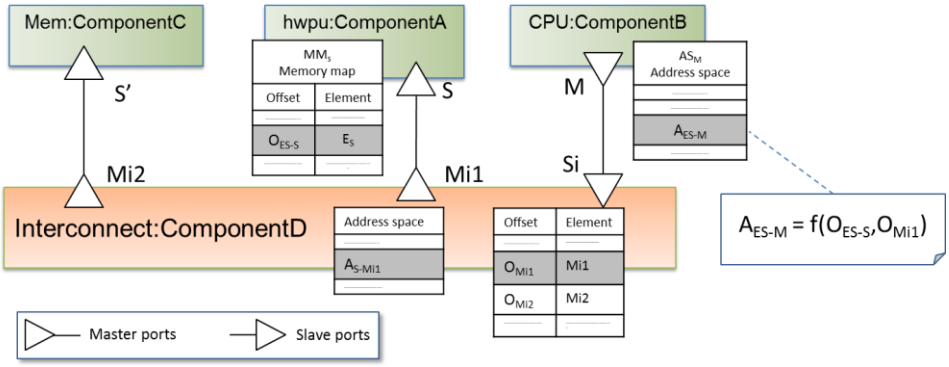


Figure 1: Master and slaves ports organization of a data bus

A processor has at least a master port for connecting it to the rest of the platform. The software executed by the processor typically uses read or write instructions (e.g. the "MOV" data movement instructions in assembly language) for accessing a particular storage resource. The address used to specify the location of this resource refers to the processor's master port address space, and is defined according to the address resolution mechanism.

2.1.2 The software concepts

By "software" people usually refers to a sequence of instructions allowing a processor to perform the functions of an application. By extension it may refer to a binary file, which will be effectively fetched by the processor for these instructions or to the source code as well. Nevertheless it is only through interactions with hardware resources that the software will be able to realize the expected functions.

In the commonly used programming languages, the description of software resources is based on the concepts of "variable" and "function" (or "subprogram"). The former specifies a storage resource (e.g. for data) while the latter defines a reusable piece of code intended to provide a specific service. The result generated by the execution of this service may depend on the values of one or more parameters, which are a special kind of variables.

Software modules are consistent sets of functions and variables. Interactions between software modules are based on the services they are able to provide, i.e. on the functions they make available to other software modules. An operation

[1]The exact designations may change form one standard to another

characterizes a service that can be requested. This is the main concept used for specifying software interfaces. The concept of "call" is used to designate an instruction that is a request for executing of a service. A call can be "synchronous" or "asynchronous" depending on its effect on the instruction sequence. The call is said "synchronous" when, upon this call, the current instruction sequence stops, waiting for the execution of the requested service and then restart at the next instruction once this execution ends. With an "asynchronous" call, the request is still sent, but the calling sequence of instructions does not wait and continues with the next instructions, in order. By the way, only synchronous calls can provide return values.

2.1.3 Common concepts

To improve productivity and clarity, both high level programming and modeling languages replace the basic concepts of bare hardware components and assembly languages by more or less advanced constructs. Raising the level of abstraction is desirable in order to decouple the functional aspect, related to the need which is usually stable, from the implementation that is linked to an evolving technology. For a developer dealing with the low level interface between hardware and software, the issue is to find among these advanced constructs those that can provide a straightforward and semantically consistent description of the technology actually available at this level.

Even if the formalisms and the vocabulary used to describe hardware and high level software may seems quite different, conceptually they are not. Indeed, and despite they map to different abstractions, both domains are built on the concepts of storage and service. This convergence is imperative because, as pointed out in §2.1.2, any software code, whatever the programming language used, will be finally executed by the interactions of hardware components.

However, due to their nature, hardware and software interfaces do not have the same flexibility. While the interfaces of software components can easily be explicit and usually follow the "one service one operation" rule, the mapping between the interface of the hardware components and the services they provide can hardly be that straightforward.

For any hardware component, invocating a service is realized by writing one or more values at a specific storage location and, depending on the kind of service, sometimes reading one or more values to get a result. Read and write instructions, such as those based on the well know "MOV" assembly code, are the basic services for interacting with hardware components. These instructions can be modeled by SysML operations which will be invocated using synchronous calls. Indeed, even if the services provided by the hardware component are usually executed asynchronously, the invocations of each basic read or write instruction is actually a synchronous call. By the way, any processor instruction can be mapped to a SysML::Operation. Since the software is executed by the processor, the interface between the software and the hardware is definitively the instructions set of the processor.

Interruptions are another interaction mechanism used by hardware components. They can be efficiently modeled by Signal events which correspond to asynchronous calls. If a software component intends to react to a specific hardware interruption, it has to provide a piece of code specified as the handler for this interruption. In SysML this handler maps to a Reception.

Both grounded on SysML::Operation and SysML::Reception concepts, software and hardware interfaces becomes seamless and easy to integrate in the same model. Of course, interacting with hardware component remains more crude and complex but it is precisely the role of the hardware abstraction layer to deal with this.

3. INTERFACE PRINCIPLES

3.1 GENERAL PRINCIPLES

In this paragraph we list the possible interactions with a hardware component, through its hardware interface, using SysML::Operations and SysML::Reception. In addition some practical constraints that characterize industrial projects are considered:

- The common practice is to use symbolic names for identifying the storage spaces
- The number of addressable storage spaces per component may be huge
- No assumption can be made on the mapping between the symbolic name and the actual storage location which may change along the development phase but which shall be resolved as a specific value (i.e. an address in the memory map) at the end.

Relying on an abstraction layer, the proposed approach provides: on one side a flatten view of the services offered by addressable hardware components and on the other side the low level interface to the bare hardware platform.

3.2 INTERFACE FEATURES

3.2.1 Operation-based hardware interface

The service-based hardware interface provided upon the abstraction appears as a list of services published (i.e. offered) by the hardware platform. These services are modeled by SysML::Operations. They may be organized on a per component basis.

The names of these operations describe the services they are related to, hiding the registers actually involved. For example, operations like: *start(), stop(), getSpeed(), setSpeed()* or *doSoftReset()* will be provided rather than: *writeStartBit()* or

readStatusRegister(). In the same way, the types of the operation parameters come from the functional domain of the corresponding hardware component, abstracting the conventions used by the implementation regarding the values stored and the precise location. Then, if a *setSpeed()* service works with a restricted set of available baud rates, an enumeration will be used (e.g. with the literals '56600' and'115200') and not the corresponding values of the bit-fields (e.g. '011' and '100') that will be written in the control register at the end.

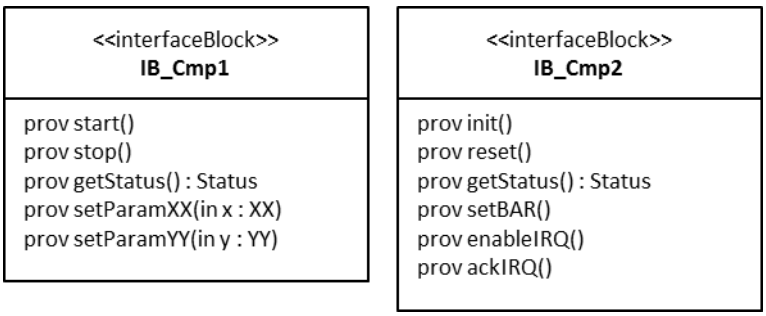


Figure 2: Examples of Hardware Abstraction Level APIs

These services, together with the protocol descriptions and the timing constraints, are sufficient to specify how the hardware platform can be used by a software component, assuming the HAL is provided. They can be specified at an early stage of the development process. The details of the implementation, which includes the coding conventions used, the hardware platform topology and any change thereof will be managed within the HAL and will not be propagated to the software components.

3.2.2 Hardware low-level interface

The interface between the HAL and the bare hardware platform is abstracted according to the approach described in §2.1.3. Three kinds of services are required:

- "write" services are modeled as SysML::Operations provided by the hardware. They change a value stored at a specific location within the memory map. A "write" service has two input parameters: one for the location and one for the value to store in it. It has neither output nor return parameter.
- "read" services are modeled as SysML::Operations provided by the hardware. They retrieve a value stored at a specific location within the memory map. A "read" service has one input parameter for specifying the location. It has one return parameter, which will be assigned the requested value, and no output parameter.
- "exception" services are modeled as SysML::Receptions provided by the HAL. They specify handler for the exceptions mapped to the corresponding signals. Just like processors exceptions, they have no parameter.

Since hardware platforms usually support more than one data type (e.g.: bit, byte, word, word32…), specific "write" and "read" services will have to be defined.

3.2.1 Abstracting memory maps and address spaces

The specification of the services can use symbolic values instead of real addresses to refer to locations of specific hardware storage resources (registers or bit-fields) in the memory map. This is a way to decouple the development of the HAL from some changes of the hardware design. Since the memory map highly depends on the hardware topology, it increases the co-design capabilities. The symbolic values will be replaced by the actual addresses on the final platform once fully specified. This implies that any element the memory map refers to can be uniquely identified.

3.3 PROTOCOLS

For a resource offering a set of services, it may be some constraints in the way these services have to be used for working as expected. For instance, some components need to be initialized or parameterized before providing any other services. In addition there are resources that cannot be reached directly. Accessing resources may become tricky when the topology of the hardware platform is complex because of the interleaving of protocols.

When a hardware resource is directly accessible from the processor executing the software (i.e. no interconnect is used), the developer has to deal with the protocol of that resource only. Unfortunately, in most cases, those resources are reachable only through one (or more) interconnects, among which some may be opaque. In such a topology, accessing the resource implies interweaving the protocol of that resource with the one of the opaque interconnect that relates it to the processor. For transparent interconnects, there is no need for taking care of the protocol since resources behind them are directly mapped into masters address spaces. The stacking of protocols significantly increases the complexity and may turn to a real nightmare when several interconnects are cascaded.

The discussion on the way SysML can be used to model protocols would be worth investigating. The main problem is that the SysML::InterfaceBlock provided in place of the UML::Interface metaclass has no feature for this. Could the pre and post-conditions together with invariants sufficient for describing the protocols of hardware resources? This is an interesting open point but it is out of the scope of this paper. In our context, we will assume that the protocols are specified, whatever the way.

4. THE HAL MODELING PATTERN

The goal is to encapsulate protocols for offering a set of abstract but straightforward services, i.e. an Application Programmable Interface (API) for the development of applications that will execute on the platform, whatever its topology.

To deals with this, we adopted a progressive approach. The first step is carried out independently for each addressable hardware resource on the platform, modeled as a part of the SysML::Block representing the overall HAL. Each of those parts is typed by the SysML::Block responsible for dealing with their own protocols, as specified by their datasheets. A given resource (i.e. part type) may have more than one API and, even for a given API, more than one connection point. For instance, a data bus usually provides number of connection points for both "master" and "slave" APIs. Connection points are modeled using SysML::Ports typed by the SysML::InterfaceBlock which specifies the corresponding API.

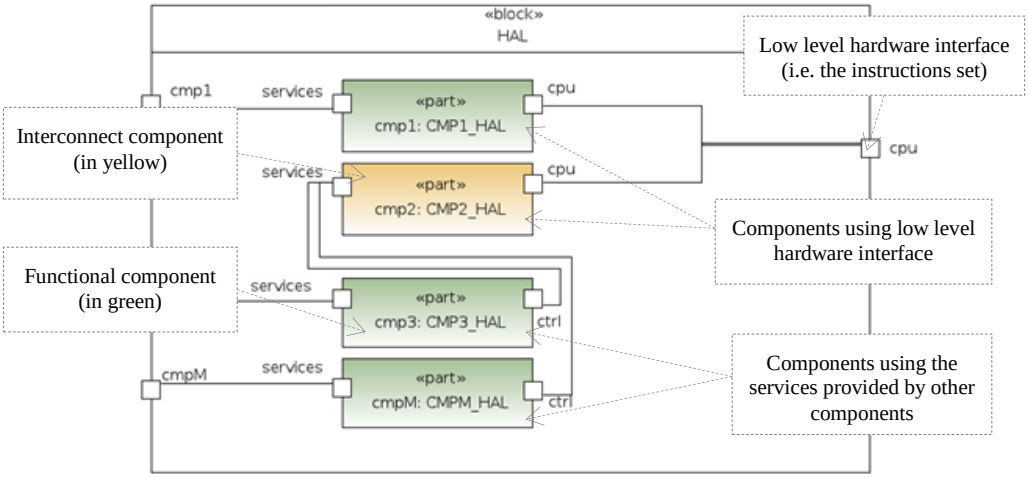


Figure 3: modeling principle applied to the Hardware Abstraction Layer

In the second step, the parts of the HAL are interconnected to mirror the topology of the hardware platform, according to the following rules. For each HAL part representing an addressable hardware device:

- If it is located on the processor's address space, it is directly connected to the HAL "cpu" port standing for the hardware platform
- Otherwise, follow the connection path from the device up to the processor:
    - For each intermediate connection, make sure the end closer to the processor is connected to the right port. In some cases, when a connection uses more than one APIs, several ports and then several connectors may be involved.

The third step consists in specifying, for each part type involved in the HAL, how the services provided by its API are realized based on services provided to it by:

- Either the API of the HAL's part it is connected to
- Or based on the CPU instruction set, if it is directly on the processor address space.

This design pattern is scalable since it actually breaks down the stack of protocols so that they are addressed one at a time, which allows dealing with topologies of any level of complexity.

5. APPLICATION TO THE STUDY CASE

Assume the hardware platform shown Figure 4. This is a very simple architecture sufficient for illustrating our modeling approach. One single central processing unit core (CPU), with its associated memory chip (Mem), drives two discrete signals (DSO) through two identical controllers. The first one, controlling DSO_1, is directly connected to the processor's bus. The second one, controlling DSO_2, is located behind a bridge (Z-Bridge).

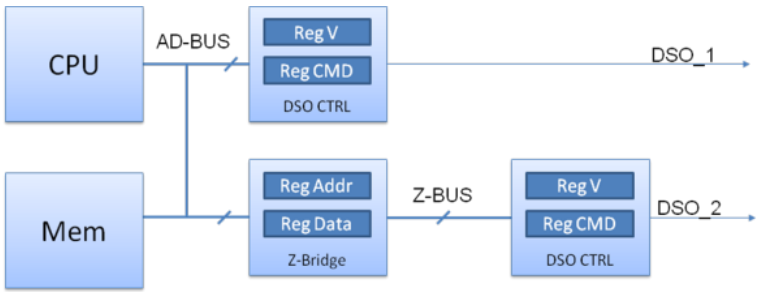


Figure 4: Study case description

As a study case, we will describe how to specify an API allowing an application to set the value of any of the DSO using a simple *set()* operation, independently of the topology and on the local protocol of the DSO controller.

5.1 INTERFACES MODELING

Regarding the processor unit, according to the proposed approach and considering that we focus on using the platform resources, we only need a limited number of behavioral features: operations for read and write and signals for exceptions. In practice, read and write operations should be overloaded to deal with all the data formats supported by the processor. In the context of this study case, we assume that there is only one format (32 bits data words), and that no burst is available. Similarly, we limit to three the number of exceptions managed: one for starting the normal execution ("main" signal), one for the "divide by zero" exception ("div0" signal) and the last one for the hardware exceptions ("irq" signal). The modeled interface is depicted Figure 5.

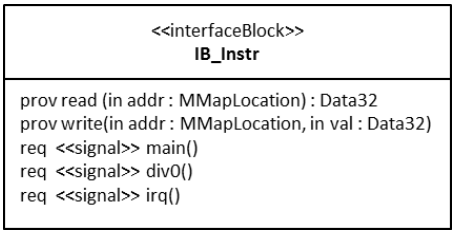
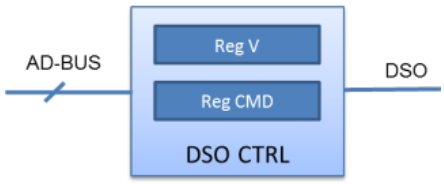


Figure 5: CPU Interface modeled using a SysML::InterfaceBlock, as exposed by the HW platform

Note that, assuming this InterfaceBlock is used to type the port on the platform side and according to the SysML convention, the read and write operation are declared "provided" while the signal handlers for the three exceptions are declared "required" (i.e. it is up to the software application to provide these handlers). On the HAL side, the port will have the same type but it will be conjugated (i.e. isConjugated=true).

The DSO controller used is imaginary but it has a representative interface. Its datasheet is presented Figure 6.

Memory Map				
Register	**Size**	**Offset**	**Access**	
RegCMD	8bits	00	R/W	
RegV	8bits	01	R/W	
Bitfield	**Register**	**Size**	**Access**	
Start/Stop	RegCMD[0]	1	R/W	
DSOVal	RegV[2]	1	R/W	

Programming interface	
Start/Stop	If this bit is set to 1 then the controller drives the output with the value set in RegV. If this bit is set to 0, then the output is in high impedance state.
DSOVal	If the controller is started, this bit controls the output state: a 1 in this bit drives the output at high state; a 0 drives the output at low state.

Figure 6: DSO controller datasheet, as used for the study case

The API we want to provide for the DSO controller is shown Figure 7. It offers the following services:

- Start/Stop services are provided through the corresponding `start()` and `stop()` operations. They have no parameter.
- An `isStarted()` service returns the Boolean value "true" whether the controller is started and "false" otherwise.
- The `setValue(in val : DSOState)` service sets the value of the controlled DSO according to the value of its input parameter which is of a discrete typed specify by the DSOState enumeration.

- The `getValue():DSOState` service returns the value actually reflected by the DSO at the time it is called

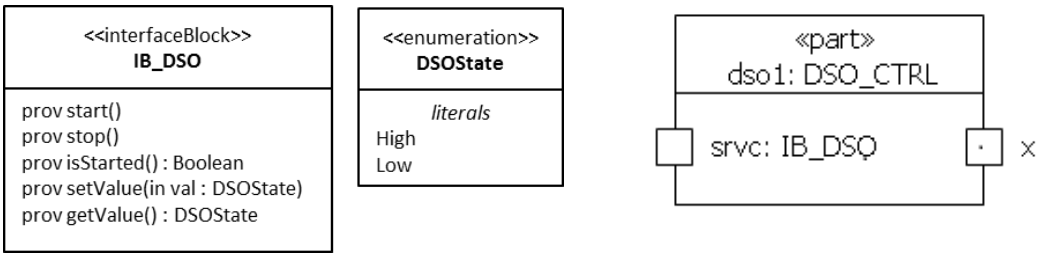
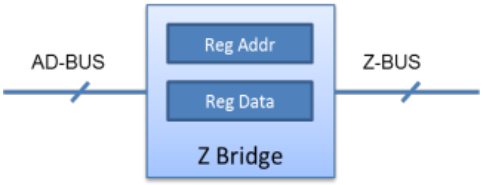


Figure 7: API provided for the DSO controller on the application side of the HAL

In the same spirit, we assume the datasheet described Figure 8 for our "Z-Bridge" controller.

Memory Map

Register	Size	Offset	Access
RegAddr	32 bits	00	R/W
RegData	32 bits	04	R/W



Programming interface

- A read access to the RegData triggers a read access on the Z bus, at the address stored in the RegAddr, and returns the result.
- A write access on the RegData triggers a write access on the Z bus at the address stored in the RegAddr register.
- The Z bus has a 20 bits address bus only. Only the 20 lowest significant bits are taken into account to locate an access on the Z bus.
- For any access to a component register through RegData register of the Z bus, its address must be set in the Z bus RegAddr register first.

Figure 8: Z-Bus controller datasheet, as used for the study case

The API we want to provide for the Z-Bridge is shown Figure 9. It offers the following services:

- The `setData(in addr : MMapLocation, in val : Data32)` service allows setting the value of a register of a component connected to the bus. The register is identified by its location in the memory map, as provided by the "addr" parameter, and the value to assign is provided by the "val" parameter.
- The `getData(in addr : MMapLocation) : Data32` service returns the value of a 32 bits register at the memory map location provided by its "addr" parameter.

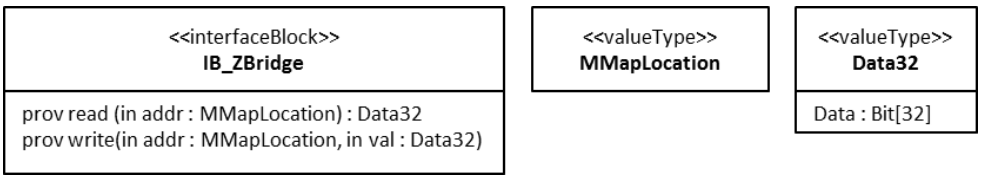


Figure 9: API provided for the Z-Bridge on the application side of the HAL

On the hardware interface side of the HAL, there is only one connection point to the CPU which will execute the software the HAL is made of. At this point, interactions are restricted by the processor instruction set. This is reflected by the IB_Instr InterfaceBlock typing the hardware platform port used by the HAL. The DSOs provided by the hardware platform are represented by two distinct ports type by the IB_DSO_Digital InterfaceBlock. The description of this last port type is out of the scope of this paper.

The model of the resulting architecture is shown in Figure 10. Assuming the protocols related to the HAL API are provided, it suitable for the development of the software application.

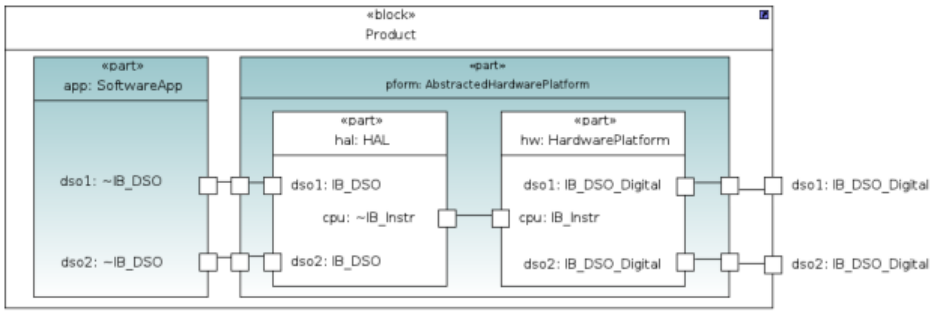


Figure 10: Resulting architecture for the study case, as used for the development of a software application

5.2 ABSTRACTION IMPLEMENTATION

Once the digital architecture is available, the internal specification of the abstraction layer can be completed, according to the approach presented in §4. Figure 11 shows the resulting model.

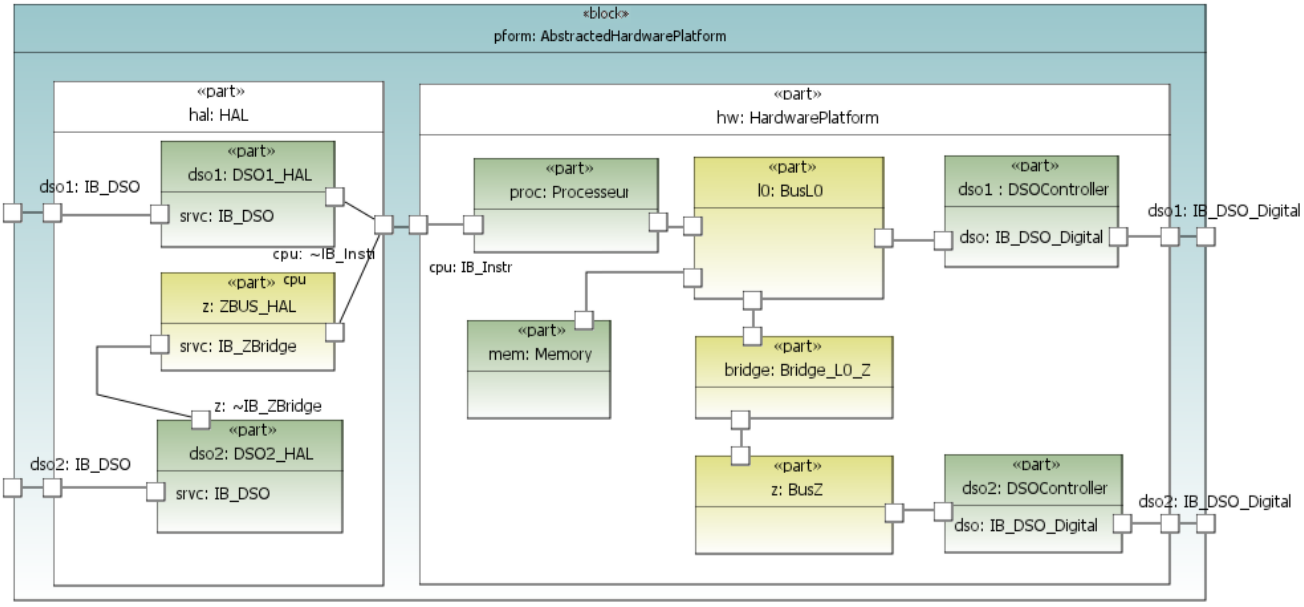


Figure 11: Digital architecture of the hardware platform and its corresponding HAL

In the hardware digital architecture, the dso1 controller and the z bridge component, which are connected to the processor's bus, are directly mapped in the address space of the processor's master port. By the way, in the HAL, their corresponding abstractions use the processor instruction set directly, as modeled by their "cpu" ports.

The gap between the services offered at the API level and those actually provided by the processor instructions has to be filled by the behavior (i.e. the algorithm) specified for each of the operations provided by the HAL to the software application. Theoretically, any kind of behavior description may be used but, in practice, SysML::Activity graphs and SysML::OpaqueBehavior encapsulating pseudo-code or executable language are the best solutions. Alternatively, the description of the expected behavior can be limited to the specification of pre and post conditions together with a list of invariants, if any.

For the purpose of our study case we chose an imperative pseudo-code. The API services provided by the HAL for the DSO1 controller are specified as follow:

```
start() {//set hw::dso1::regCMD bit 0 to 1
      write(hw::dso1::regCMD,read(hw::dso1::regCMD) or 0x01);}

stop() {//set hw::dso1::regCMD bit 0 to 0
      write(hw::dso1::regCMD,read(hw::dso1::regCMD) and 0xFE);}

isStarted() : Boolean {//test  hw::dso1::regCMD bit 0 value
      if(cpu.read(hw::dso1::regCMD) and 0x01)
            return True;
      else
            return False;}

setValue(in val : DSOState) {//set hw::dso1::regV bit 2
      if(val==DSOState::High)
            cpu.write(hw::dso1::regCMD,cpu.read(hw::dso1::regCMD) or 0x04);
```

```
	else
		cpu.write(hw::dso1::regCMD,cpu.read(hw::dso1::regCMD) and 0xFB);}

getValue() : DSOState {//test hw::dso1::regV bit 2
	if((cpu.read(hw::dso1::regV) and 0xFB)!=0)
		return DSOState::High;
	else
		return DSOState::Low;}
```

While the API services for the Z-Bridge are specified as follow:

```
setData(in addr : MMapLocation, in val : Data32) {
	cpu.write(hw::bridge::RegAddr, addr.toData32());
	cpu.write(hw::bridge::RegData, val);}

getData(in addr : MMapLocation) : Data32 {
	cpu.write(hw::bridge::RegAddr, addr.toData32());
	returncpu.read(hw::bridge::RegData);}
```

Driving the DSO2 output from the software application requires interleaving both the DSO controller protocol and the Z-Bridge protocol. However, this work can be simplified since we can rely on the bridge's service API developed at the previous step, as described below.

```
start() {//set hw::dso2::regCMD bit 0 to 1
	z.setData(hw::dso2::regCMD, z.getData(hw::dso2::regCMD) or 0x01);}

stop() {//set hw::dso2::regCMD bit 0 to 0
	z.setData(hw::dso2::regCMD, z.getData(hw::dso2::regCMD) or 0xFB);}

isStarted() : Boolean {//test hw::dso2::regCMD bit 0 value
	if(s.getData(hw::dso2::regCMD) and 0x01)
		return True;
	else
		return False;}

setValue(in val : DSOState) {//set hw::dso2::regV bit 2
	if(val==DSOState::High)
		z.setData(hw::dso2::regCMD, z.getData(hw::dso2::regCMD) or 0x04);
	else
		z.setData(hw::dso2::regCMD, z.getData(hw::dso2::regCMD) and 0xFB);}

getValue() : DSOState {//test hw::dso2::regV bit 2
	if((z.getData(hw::dso2::regV) and 0xFB)!=0)
		return DSOState::High;
	else
		return DSOState::Low;}
```

Comparing the implementation of the API for DSO1 and DSO2, we notice that the level of complexity is exactly the same, despite the additional protocol we have to deal with.

6. CONCLUSION

Using abstractions coming from the software domain, we described how we can build a low level model for the specification of interfaces to bare hardware platforms. Thanks to its level of abstraction together with the concepts on which it is founded, this approach provides a seamless transition between the hardware and the software domain. This makes it convenient for supporting model-based development of the Hardware Abstraction Layer (HAL) which hides the actual topology of a hardware platform and presents it as a flat list of services.

The approach described in this paper is also compatible and consistent with the technologies of virtual prototyping like SystemC/TLM, which enable early verification and validation. Indeed, the seamless specification model provided might be used to generate the executable model of the virtual prototype, assuming the hardware behavior is modeled as well.

Both HAL and virtual prototyping are key features for providing real co-design facilities for hardware and software components of a product

Our next step will be to investigate how SysML can be used for specifying protocols. This may require some enhancement of the current standard. Then we will study in more details how the model of the hardware/software interface can be properly related to the modeling of hardware behavior. The multiprocessors/multi-cores aspects will be another axe of our future works as well.

Acknowledgements

The authors would like to thank Airbus Operations SAS for having funded this work.

7. REFERENCES TO EXTERNAL PUBLICATIONS

[1] R. Domer, A. Gerstlauer and W. Muller, "Introduction to Hardware-dependent Software design," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, 2009.

[2] A. Bouchhima, X. Chen, F. Pétrot, W. O. Cesario and A. A. Jerraya, "A unified HW/SW interface model to remove discontinuities between HW and SW design," in *Proceedings of the 5th ACM international conference on Embedded software*, New York, NY, USA, 2005.

[3] V. Lefftz, J. Bertrand, H. Casse, C. Clienti, P. Coussy, L. Maillet-Contoz, P. Mercier, P. Moreau, L. Pierre and E. Vaumorin, "A design flow for critical embedded systems," in *Industrial Embedded Systems (SIES), 2010 International Symposium on*, 2010.

[4] P. Gerin, H. Shen, A. Chureau, A. Bouchhima and A. Jerraya, "Flexible and executable hardware/software interface modeling for multiprocessor SoC design using SystemC," in *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, 2007.

[5] A. A. Jerraya, A. Bouchhima and F. Pétrot, "Programming models and HW-SW interfaces abstraction for multi-processor SoC," in *Proceedings of the 43rd annual Design Automation Conference*, 2006.

[6] M. Baklouti, A. Benzina, A. Bouchhima and F. Pétrot, "Extending transaction level modeling for embedded software design and validation," in *Design \& Technology of Integrated Systems in Nanoscale Era, 2007. DTIS. International Conference on*, 2007.

[7] W. a. H. D. a. M. F. a. W. A. a. W. P. a. P. P. a. V. E. a. M. N. a. K. D. a. A. F. a. C. M. Muller, "The SATURN Approach to SysML-Based HW/SW Codesign," *Embedded Systems Codesign*, oct 2011.

[8] OMG, "OMG Unified Modeling Language (OMG UML), Superstructure V2.4.1," Aug 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4/Superstructure.

[9] OMG, "UML Profile for MARTE: Modeling and Analysis of Real-Time Systems - Version 1.1," June 2011. [Online]. Available: http://www.omg.org/spec/MARTE/1.1/.

[10] T. Robert and V. Perrier, "A UML design flow aimed at embedded systems," 2010.

[11] A. S. I. R. Quadri and L. S. Indrusiak., "MADES: a SysML/MARTE high level methodology for real-time and embedded systems," in *European Congress on Embedded Real-Time Software ERTS, Toulouse, France, 01/02/2012-03/02/2012*, http://www.erts2012.org/, 2012.

[12] A. Abdallah, A. Gamatié and J.-L. Dekeyser, "Modelisation UML/MARTE de SoC et analyse temporelle basee sur l'approche synchrone," *RSTI - TSI - 30/2011. Architecture des ordinateurs*, vol. 30, pp. 1089-1114, 2011.

[13] A. Koudri, J. Champeau, D. Aulagnier and D. Vojtisek, "Processus MOPCOM pour SoC/SoPC," *Génie Logiciel - Ingéniérie dirigée par les modèles*, p. xxx, 2009.

[14] IEEE, *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP wiwith Tool Flows.*

[15] IEEE Computer Society, "IEEE Standard for Standard SystemC Language Reference Manual," 2012. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6134619.

[16] OMG, "OMG Systems Modeling Language (OMG SysML) V1.3," Aug 2012. [Online]. Available: http://www.omg.org/spec/SysML/1.3/.