Standards for System-Level Design: Practical Reality or Solution in Search of a Question?

Christopher K. Lennard, Cadence Patrick Schaumont, IMEC Gjalt de Jong, Alcatel
Anssi Haverinen, Nokia Pete Hardee, CoWare
Members of: System-Level Design and On-Chip Bus Development Working Groups of VSIA

Abstract:
We address the issue of standards development for the system-level design space. System-level design IP re-use standards are key to the future of the VSIA. However, the concept of system-level standards has its share of sceptics: what role can standards play in this developing market segment? In response we present an overview of three standards in the system-level VC integration space, and describe two distinct industrial case studies to support their practicality.

1 Introduction

Three factors drive the need for standards within the design and EDA industry. These are the need for: (1) common communication principles, (2) common design formats, and (3) a unified approach to design-quality measurement and assurance. Standardising communication permits straightforward connection of tools (e.g., via APIs), virtual components (VC) (e.g., via bus standards), or complete SoC designs (e.g., telecom standards). Standardisation of design formats ensures comprehensive design-property encapsulation within a small set of alternatives (e.g., RTL synthesizable subset of VHDL or Verilog). Finally, quality-based standards must help resolve an industry-wide design-quality issue. For the VSIA, the quality issue is the guarantee of SoC design integrity while ensuring fast integration of VCs from multiple sources.

To ensure "design quality" in the rapidly developing system-level design space, standards must help to align existing technology and design principles with emerging concepts. They link the fundamental principles developed in academia to the industrial needs of: a) the handling legacy-components, b) catering to customer risk-tolerance, and c) understanding how the flurry of new system-level tools enhance existing design flows.

The VSIA is producing standards to help the industry take small-steps towards comprehensive adoption of system-level architectural exploration and VC exchange [1]. To achieve this, the VSIA is standardising a definition for "interface-based design" at all levels of design abstraction. At the system-level, this implies common VC description techniques ensured through a unified taxonomy ([2], [3]) and well defined separation of VC interface (protocol) and the VC internal behaviour. This standardisation of "mutual comprehension" is a quality assurance of documentation for VC hand-off. The VSIA must also develop "interoperability" standards to simplify integration overhead today. This includes the standardisation of communication principles – for example, the On-Chip Bus VC interface; and standardisation of design-format features – for example, design-language data-types.

In this paper we describe three emerging VSIA system-level integration standards which are already gaining industrial adoption. These are the System-Level Interface Behavioural Documentation (SLIF) Standard, the On-Chip Bus Virtual Component Interface (OCB VCI), and the System-Level Data-Types standard. The first of these is a mutual comprehension standard for rigorous interface-based description of any VC, legacy or new. It enforces a system-level view upon standard VC integration, and provides the link between abstract models and VC implementation. The latter two standards are interoperability standards, and both tie in with use of the SLIF standard. The OCB VCI transaction-level view provides a bus-interface abstraction, which is not limited by the VC integrator's choice of bus. The OCB VCI transactions are the first step towards full interoperability standards for VC interfaces across multiple system-design abstractions. The standard data-types permit quick analysis of interoperability requirements, and guarantees that a common interpretation of data-operations is used within the VC behaviours.

2 The VSIA Interface Standards

2.1 The Interface Based Design Principle

To enable mutual comprehension and interoperability, system-level VCs need clean separation of internal-behaviour and protocol ([4], [5], [6], [7]). All VC documentation (and eventually models) must specify internal behaviour independent of protocol, and a protocol block independent of internal behavioural details.

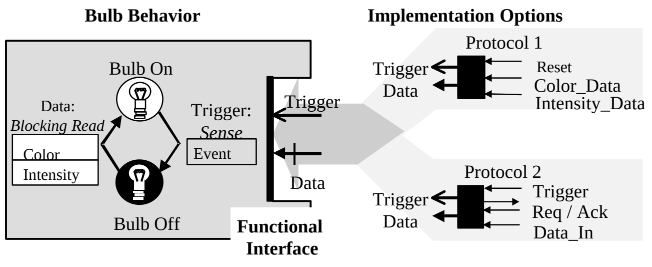


Figure 1. *The Separation of Behaviour and Interface*

As an example an interface-behaviour separation is shown in Figure 1. The light-bulb behaviour on the left-hand side (LHS) is a simple ON/OFF action. The ON action requires a basic communication task: Read "*Colour*" and "*Intensity*". The arrival order and format of this information is irrelevant to the conceptual behaviour. The arrival order and format of the data is, however, critical to each instantiation of the VC. In this example, we have suggested two protocol instantiations on the right-hand side (RHS). These both satisfy the single behavioural communication requirement of the VC whilst separately encapsulating the implementation requirements of a protocol.

The internal behaviour and the protocol block are separated by the Functional (or *Layer 1.0*) Interface. This Functional Interface supports a limited and well-defined legal transaction set to ensure a common definition of behaviour-interface separation. The restrictions upon the Functional Interface are:

a) Actions on ports are not explicitly coupled or related (e.g., no handshaking)
b) Any sequencing actions on these ports is a property of the internal behaviour
c) The only allowable transactions are: Read, Write, Sense and Emit
d) Data-types passing across the interface can be of arbitrary complexity

To bridge the abstraction gap between the Functional Interface and protocol block output, a hierarchical interface-refinement strategy is supported. The protocol block is be described as a series of refinement layers taking the Functional Interface (Layer 1.0) properties down to implemented protocol (Layer 0.0).

Deriving a clean interface-behavioural separation satisfies the hand-off and integration principles of *mutual comprehension* and *model interoperability*. Mutual comprehension is supported by the classification of communication properties and protocols into standard semantics and abstraction hierarchies. Model interoperability is achieved in two ways. First, it allows the VC integrator to relate the communication principles of the behaviour to the details of a protocol implementation. Secondly, the separation of the interface permits simple linking of a VC's behaviour into standard communication protocols (e.g., APIs)

2.2 The Standards

2.2.1 The System-Level Interface Standard

The VSIA System-Level Interface Behavioural Documentation Standard (SLIF) achieves uniformity of hierarchical description for VC interfaces. The interface description standard follows the interface separation principle outlined above. It possesses a sufficient set of interface transaction and message classes for the description of any interface abstraction layer. Each of these transactions and messages can support attributes to express communication intent. Diagrammatic shorthand for representation of the interface layers, including ports, transactions and attributes is given. This notation indicates data producer/consumer and action initiator/responder relationships. Example transactions, attributes and shorthand are shown in the Table 1.

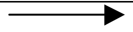
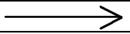
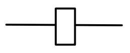
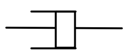
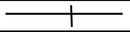
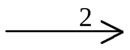
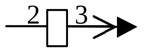
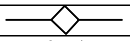
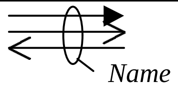
Transactions	**Attributes**	**Notation**
messSense	Data Flow	
messEmit	Control Flow	
transRead	Buffer / Persistence	
transWrite	FIFO	
transOpenChannel	Blocking	
transCloseChannel	Priority	
transSynchronize	Multirate	
transReset	Pipelined	
transControl	Protocol Related	

Table 1. *SLIF Transactions and Attributes*

To ensure industry-standard separation of interface and internal behaviour in a VC description, a restricted set of transactions and attributes are permitted at the Functional Interface. Explicit temporal association between transactions is also not permitted at the Functional Interface. Documenting the basic communication principles of each VC through use of the Functional Interface is mandatory. This makes the most abstract interface layer computation-domain neutral.

The SLIF Standard is a hierarchical interface specification with consistent content requirements, section numbering, and port / transaction naming strategy for each interface layer. The format ensures completeness and clear identification of the structural and transaction relationship between layers. The hierarchy permits verification of interface-property inheritance on a layer-by-layer basis.

For situations where a VC connects to a bus, this documentation approach can be linked into the interoperability standards of the OCB VCI. However, the SLIF standard can be able to be applied to any form of VC or VC model whether bus-based or not (e.g., dedicated hardware components, cores, SW objects, etc.). The SLD DWG is supporting multiple VC delivery models with this standard. Each interface layer must have supporting documentation, but it is optional to supply a simulation model or implemented VC object for each abstraction. In this way, the documentation standard can be applied directly to legacy VC as just a clarifying measure.

2.2.2 The OCB VCI Standard

The OCB Virtual Component Interface standard defines a generic cycle-based address-mapped point-to-point interface rather than a bus. It does not demand nor define particular bus allocation schemes or bus protocols. Instead, the OCB VCI provides a set of logical signals with a flexible and extendable protocol to transfer information between two end points, (e.g.) a VC and the interface logic of a bus. (See Figure 2). The OCB VCI standard specifies:

1. a request-response transaction protocol
2. a protocol for the transfer of those requests and responses,
3. the contents and coding of these requests and responses.

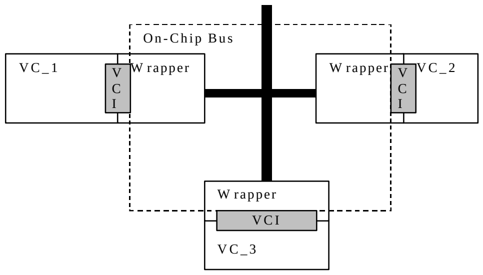


Figure 2. *Use of VCI with an On-Chip Bus*

The VCI will generally require a “wrapper” between the interface and the bus. This wrapper may be specific translation logic or a parameter-driven generator of translation logic.

The advantage of using the VCI is that it greatly improves integration across multiple platforms. To support true mix-and-match of VCs, the VC provider should not need to know the system interconnect. A VC provider can now design to the VCI as a single interface (as shown on the RHS of Figure 3), and the integrator understands how this is translated to their OCB standard. Tuning the VC connection to the system’s communication channel (bus) by building of the wrapper (LHS of Figure 3) is left to the system integrator.

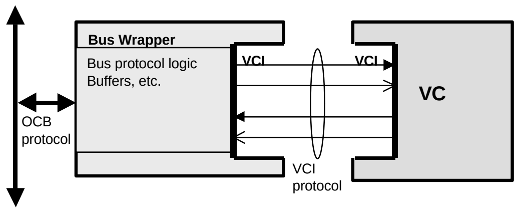


Figure 3. *Separating Bus-Specific Logic from the VC*

There are three complexity levels for the OCB VCI: Advanced VCI, Basic VCI, and Peripheral VCI. All these interfaces are compatible with each other. The Peripheral VCI is the most generic interface and it implements a simple communication channel without features to enhance performance. The Peripheral VCI is comparable to any peripheral on-chip bus interface, and its targeted use is with VCs, which connect to such buses and for generic point-to-point communication. The Basic and Advanced VCI contain features typical with high-performance on-chip buses, such as split transactions, pipelining, threads, packets. Thus, a component with an Advanced or Basic VCI can be connected to a high-performance system bus with minimal wrapper logic.

Although the OCB VCI standard has mechanisms for building VCs with complex high-performance interfaces for system-bus connectivity, the predominant use of the standard by VC providers is expected to be the simple Peripheral VCI.

A further definition of the OCB VCI standard is a transaction language format. This can be used to define simulation vectors for VCI compliance tests, but also provides the link into the SLIF standard. This transaction language provides a higher abstraction for the communication that isolates the designer of the test cases from interface signals, clock, and protocol in general. The example of Figure 4 shows how an abstract transaction maps to Peripheral VCI signals:

```
vciStore (0x3, 0xF, 0x12);
```

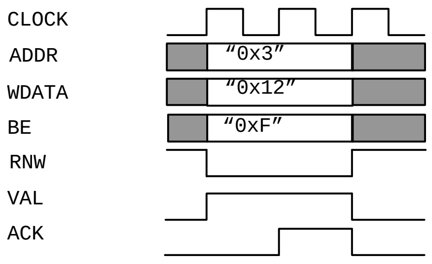


Figure 4. *An Example of VCI Transactions*

2.2.3 The System-Level Data-Types Standard

This data-type standard addresses the issue of highly variable system-level data-types and operations. This variability in the industry has made the portability of software VCs and simulation models difficult. The solution offered by the VSIA Data-Types standard is one of standard header-file syntax and data-type semantics for C/C++-based data-types. The goal is to achieve a result similar to what the IEEE 1164 package has achieved for VHDL [8], [9]. The first version of the standard will cover built-in types for: int, short and the IEEE standards for floats; and will also address the bit-true versions of 'std types': bitvector: uninterpreted, signed and unsigned integral types, and signed and unsigned fixed-point types.

Standardisation of data-types is recognised as critical in any system-level C/C++ language. Example language developments which have identified this need are: SystemC [http://www.systemc.org], Cynapps [http://cynapps.com] and Superlog [http://www.co-design.com]. Cleveldesign [http://www.cleveldesign.com], and Frontier Design [http://www.frontierd.com]. The VSIA is seeking data-type alignment will all these efforts.

2.2.3.1 Basic Concepts

The data types and their semantics are defined upon the following of basic concepts: no assumptions on compilers and run-time environments; explicit conversion functions to avoid implicit conversions; all types have a string representation to serve as foreign exchange mechanism thereby eliminating reliance upon `stdio` or IO stream class libraries; defined initial values for all types; and implementations to have freedom with respect to data representation, additional C++ keywords, type inheritance, analysis capabilities, etc.. Further, must be the development of a test-bench set so that vendor compliance can be measured.

For consistency between all related types and their semantics, a user centric use model is applied. This provides a rich set of functions and operators. For performance, good coding style of the defined data types is demanded. We classify the set of types into four classes: base-types, bit-vectors, signed/unsigned numeric types, and fixpoint types

2.2.3.2 Base types

Eight standard types have been defined: `vsi_int`*n* and `vsi_uint`*n* where *n* can have the values 8, 16, 32 and 64. This reflects common use and best practice of a set of typedefs to fix the size of the native char, short, int and long types, which are machine dependent in ANSI C.

Other supporting types are: `vsi_base` which is an enumeration type describing which base format, e.g. binary, octal, decimal, or hexadecimal, used in the string representations. `Vsi_overflow_mode` summarizes all defined overflow mode characteristics used in assignments. Wrap-around and two saturation modes are defined. For the fixpoint types, quantization modes will be defined.

Objects of the `vsi_context` class are to set default and initial values for lengths of bitvectors at construction time and overflow mode characteristics.

The `vsi_bit` type is a full replacement of the '`bool`' type. This was defined to allow extension to a multi-valued logic type. Two constants, `VSI_ZERO` and `VSI_ONE` are defined. `Bool` and "`const char*`" constructors and assignment variants are defined. The equality and inequality operators are defined for all constructor variants, together with the bitwise operators `~,| &` and `^`. No ordering between the "zero" and "one" value exists. .

2.2.3.3 Vsi_bitvector

The `vsi_bitvector` type is a true bitvector type, with a non-zero, positive size. `int` and `const char*` constructor are defined, where the `int` constructor defines the length. String conversion to and from the bitvector type are defined; variants with and without base are defined. Assignment operators, including shifts, are defined for all constructor variants. Only the equality and inequality operators are defined for all constructor variants. No numerical interpretation and operations are defined, only logical bit-wise operations `~,| &` and `^.` The left and right shift operators are logical shifts. A general shift function and rotate functions are also included. The size of an object does not change during its lifetime, regardless of operations upon it. Reverse and concatenation operations exist as additional utility functions.

Index and subvector operators can also be used as lvalues, i.e. a vector can be assigned and partially modified. For example, expressions like `vec1[7] = vec2[7]` and `vec1(3,0) = vec2(7,4)` are allowed. To achieve this behavior, proxy classes `<type>_bitref`, and `<type>_subref` have been defined. The proxy classes behave as close to the associated type. Template classes like `vsi_Tbitvector<n>` and `vsi_Tsigned<n>` are defined for user-friendliness. The size of the vectors are directly visible, and optimized implementations can be provided.

2.2.3.4 Vsi_signed and vsi_unsigned

The `vsi_signed` and `vsi_unsigned` types are true numerical types with a defined bitvector representation, which is in 2's complement, and with MSB as first element. `Int`, `(un)signed` and `const char*` constructor are defined. The int constructor determines the length of the object. Full-fledged constructors, with initial value and overflow characteristic, also exist. String conversion to and from the bitvector type are defined; explicit conversions from and to `vsi_bitvector` also exist. All relational operators are defined. The ordering, as for differently sized objects, is defined by considering the represented numerical values. All bit-wise and arithmetic operations are defined. Additionally a general shift and remainder function are included. In contrary with ANSI C, division, modulo and remainder are unambiguously defined. All operations are defined with arbitrary precision; overflow mode characteristics due to fixed sizes of the values are considered at assignment time. For the index and subvector operators, (implicit) sign extension is considered when the value is accessed "out of bounds".

2.2.3.5 Fixpoint types

The fixpoint data types will be defined as the natural extension of the integral types `vsi_signed` and `vsi_unsigned`. A fixpoint value is like an integral value, but with 2 lengths. One length is the length of the integral part; a second value is the length of the fractional part. A set of quantization modes are to be defined.

3 Applying the Standards to Industry

This section is a description of the application of the standards presented above. Three industrial case-studies are being performed by the VSIA SLD DWG as pilots for the standards. The first study is the application of the SLIF standard to the telecom design flow within IMEC. The second involves the application of the SLIF and OCB standards to VC exchange and bus integration between projects in Nokia and CoWare; and the third involves the application of all three standards on a design project within Alcatel. The first of these case studies has completed. The latter two are ongoing at the time of authoring this paper, and the Nokia / CoWare project intent and status is described following the IMEC study.

In each case, the application of these standards is broken down into: (a) the environment and problem, (b) applying the standards, and (c) the results of application.

3.1 Application within IMEC

3.1.1 Environment and Problem

At IMEC, "demonstrator" designs are being developed in the field of telecommunications and multimedia. The design flow relies on the use of C++ to capture the entire design path from system level exploration to detailed hardware design (Figure 5). The SLIF standard has been applied to assist with the following IP-related issues:

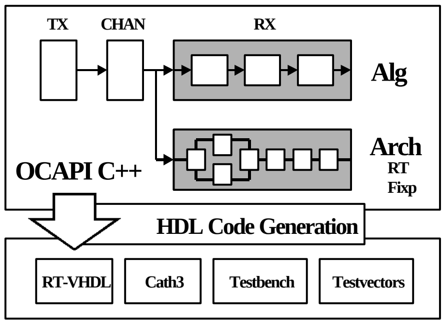


Figure 5. *C++ based design flow in use at IMEC*

1. Demonstrator-design concentrates on the novel parts of an application. The VC model capture using the interface-based design style of the SLIF standard is very effective at providing appropriate design focus.
2. The SLIF standard handles high level behavioural models. This allows easy interpretation of a system model by an external partner.
3. The SLIF documentation provides unified structure for demonstration of the results from the design methodology research at IMEC.

The driver application that was used in the pilot project was a digital gain control block for a modem. This VC, called `CMULT` performs a constant multiplication of an input value with a parameter. The parameter is also programmable. System integration of this model raises the following issues: under what conditions can the parameter be updated?; when can the input be read?; what interface signals govern consumption and production of data? The SLIF standard supports accurate documentation of these issues.The following two subsections describe how the SLIF standard helps to expose the interface protocol information so that the above issues are easily resolved.

3.1.2 Applying the Standards

The SLIF standard promotes interface-based design by using documentation *layers*. Each layer describes a VC at one level of abstraction, and each can use different execution and communication semantics. All layers fit on top of each other through refinement. This way, the initial layer expresses the most abstract view of a VC, while subsequent layers elaborate the abstract views into more concrete views.

Figure 6 shows the uppermost layer of the CMULT (constant multiplication) VC. This VC is drawn using the standard attributes of the SLIF standard. It shows two

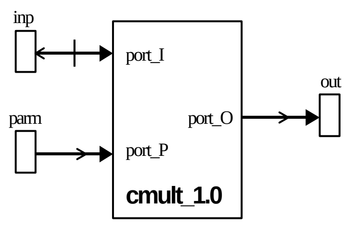


Figure 6. *CMULT Layer 1.0 External View*

input interfaces (`inp` and `parm`), and one output interface (`out`). The `inp` interface reads in data values through `port_I`, and produces values through `port_O`. Interface `parm` is used to update the parameter value to multiply with. The arrows attached to each interface explain the communication semantics that are used on the interface ports. Consider for instance `inp`. The filled arrow into this port indicates consumption of a data flow. The transfer is initiated by `inp` itself, which is indicated by a hollow arrow (a control event) going out of `inp`. The transfer of data is also blocking, which is indicated by a vertical bar across the arrows. Thus, the transaction on `inp` expresses a blocking read.

In the C++ design environment used at IMEC, the most abstract view of a VC (the algorithmic view) uses data-flow semantics. Execution of this kind of behaviour is purely driven by the availability of data. The refined view (the architecture view) uses cycle-true simulation semantics. In that case, execution of VC behaviour is governed by the presence of a clock edge. To go from the algorithmic to the architecture view, one has to devise interfaces that implement the data-flow semantics using cycle-true semantics by means of an appropriate protocol.

The 1.0 layer implementation that is associated with the SLIF document will be described in C++ using data-flow semantics. For the transaction on `inp` for instance, a C++ source code fragment is:

```
execute_vc() {
  if (data present on port I) then {
      read data value;
      do processing;
    }
    // else do nothing
   return;
   }
```

In this fragment, the code related to `port_O` and `port_P` has been left out. However, the code *implements* the specification of the `inp` interface in Figure 6. This example illustrates a key-benefit of the SLIF standard. The standard introduces an intermediate layer of documentation that is *independent* of a particular design environment. The specification in Figure 6 is generic (yet unique), while the code fragment is specific to the coding style and the C++ design environment.

3.1.3 Documentation Layering

The SLIF standard documents a VC at different levels of abstraction (layers). The layers that were used to document the constant-multiplication VC are shown in Table 2. Four different layers are listed, starting from a data-flow functional interface (Layer 1.0) down to cycle-true register-transfer (RT) model (Layer 0.0). Intermediate layers describe intermediate steps in the refinement.

Layer	Interface	Behavior	Implementation
1.0	Data-flow	Data-flow	C++
0.2	DF/RT	Data-flow	C++
0.1	RT	Data-flow	C++
0.0	RT	RT	VHDL

Table 2. *Layering of the CMULT VC*

Layer 0.2 expresses the interface characteristics of the different I/O interfaces using request/acknowledge signalling. However, the complete VC still is described as a data-flow model. The 0.2 layer thus expresses the ports that will show up in the VC architecture as a result from the refinement from data-flow to RT.

The next layer (Layer 0.1) expresses the same ports, but switches the simulation semantics of the interfaces from data-flow to cycle-true. Introducing cycle-true semantics allows the expression of the VC timing characteristics in terms of clock cycles. This is not possible in an untimed data-flow model (Layer 0.2). However, even at Layer 0.1 a behavioural view of the VC's internal functionality may still be used. A possible internal view of Layer 0.1 is shown in Figure 7. Since the internals of a VC are essentially hidden, this mapping is of course not unique; the figure rather shows one possible solution of the mapping.

Comparing to Figure 6, we see that Figure 7 encapsulates level 1.0. Each of the 1.0 interfaces is expanded to a set of interface signals. In addition, a clock signal is introduced to control time in the RT model.

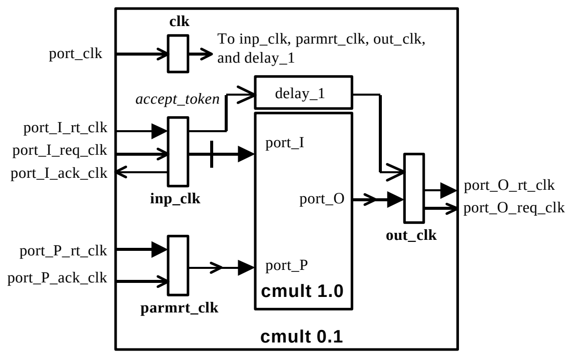


Figure 7. *Layer 0.1 Internal View*

There is also a *delay* block present. This delay block transports *time* (which cannot propagate through the cmult_1.0 data-flow model) to the output interface `out_clk`. The block is needed because, whenever the input interface `inp_clk` accepts a data value, the data-flow block cmult_1.0 evaluates an output result immediately. Yet, the cmult_0.1 model is timed (cycle true), and the output interface `out_clk` is expected to produce a result with the appropriate latency. This problem is solved with the delay block, that marks the point in time at which a new input is accepted, and that indicates when a new output should be produced.

This example illustrates that the SLIF has modelling and documentation mechanisms that support full interface based design.

3.1.4 Results of the application

From this design experience at IMEC, we conclude:

1. The SLIF standard is powerful as an environment-independent documentation standard. SLIF allows assessment of VC integration without getting lost in implementation details.

2. The SLIF standard promotes interface-based design. This is a side effect of preparing the documentation as the design flow is traversed. In the IMEC environment we have partitioned the interfaces and the behaviour as separate C++ classes and this establishes an obvious relationship with the SLIF documentation.
3. The SLIF standard allows design review process done by either SLIF experts or design environment-experts. In both cases, valuable and sensible comments were formulated that allowed improvement of the design.

3.2 Application within Nokia / CoWare

3.2.1 Environment and Problem

The Nokia/CoWare pilot project is centred on the development of components for third-generation cellular (3G) applications. The components in question are essential elements of the physical layer, also called Layer 1 (L1), of a 3G cellular system. Nokia is adopting a design methodology based on VCs and the application of the SLIF and OCB standards to VC exchange and bus integration for this project for three main reasons:

1. The 3G standards are not yet finalised and it is clear that there will be multiple variants of the standards. Known functions will therefore need to be re-used in many different design variants.
2. Some functions can be common across different sub-systems with very different system design constraints and architectures. For example, components with identical descriptions at SLIF's Layer 1.0 (Functional Interface) could be used in both base stations and handsets, with very different architectural and implementation needs.
3. It is beneficial to remain as independent as possible from the implementation, including the specifics of the processor and associated bus, until the latest possible stage.

The pilot project endeavours to show that the SLIF and OCB standards can be used, in combination with the CoWare N2C™ design system, to achieve these benefits.

3.2.2 Applying the Standards

The L1 Pilot Project Virtual Component is comprised of a convolutional encoder, Viterbi decoder and built in self-test capabilities. In the pilot project, these are implemented in a Gate Array (GA). The GA is mounted on a test board, which includes an ARM processor. The virtual component is tested with a Test Bench comprised of a rack of test equipment or via its own built in self-test capability. The Control block has a GUI in the CoWare simulation environment to allow Test setup and reporting. Each of the configurable blocks in the system can be controlled via the GUI ARM SW running on the test board will be used to control the VC's function via memory mapped I/O. The pilot project example is illustrated in Figure 8.

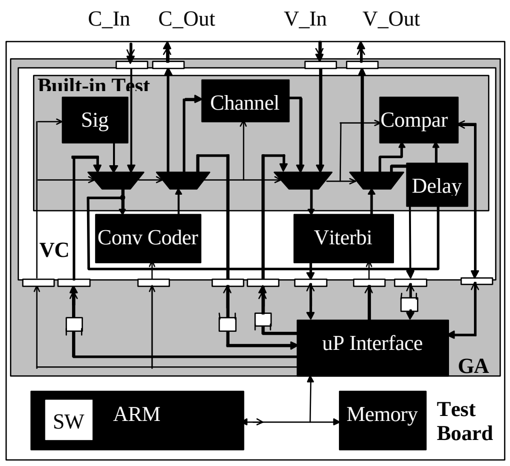


Figure 8. *Pilot Architecture – Nokia / CoWare Example*

Proving the usefulness of the SLIF method of documentation and showing how it links into the OCB specification is a key outcome for the pilot project. This is especially important since the VC would be handed off within different divisions of Nokia, so it must be understandable outside of the team that created it. Figure 8 shows the VC as implemented in the pilot project's test configuration. Only the VC portion of this diagram will be described using the SLIF. The levels of description are as shown in Figure 9.

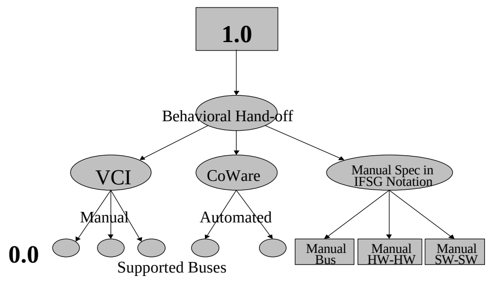


Figure 9. *Levels of Interface Description*

The behavioural hand-off level reflects some aspects of architectural choice for implementation while remaining independent of the exact implementation. From there, implementation will take place in a manner appropriate for the implementation method (i.e. manual design versus automated generation of bus interfaces in CoWare). These levels are described in parallel using the SLIF method, to check consistency of the SLIF with the other paths.

3.2.2.1 Linking Standards to Implementation

Where a VC is implemented as a peripheral attached to a bus (for the ARM processor in this case), one of the Layer 0.x SLIF layers is made to map into the OCB standard's Peripheral VCI (PVCI). From the Layer 0.x layer to the RTL (or Layer 0.0), automated interface synthesis is used. This interface synthesis step is best explained through an analysis of the similarity of the PVCI and CoWare's “Virtual Bus”.

Both the PVCI and Virtual Bus specify a generic and abstract means to define a bus interface. These interface definitions are independent of the actual on-chip bus used, thereby enabling a top-down design flow. The PVCI bus interface describes the generic signal types and protocols for requests, the responses to such requests, and contents and coding of these requests and responses. Virtual Bus also describes generic signal types and associated protocols for bus transfers. The generic categories of signals in the PVCI map closely to those in Virtual Bus. The basic PVCI handshake protocol maps to Virtual Bus' FullHndshk protocol, one of several basic protocols provided with the Virtual Bus. This pilot project will complete the mapping of signals and protocols between PVCI and the CoWare Virtual Bus.

Unlike PVCI, however, Virtual Bus is backed by tools and methodologies which enable interface synthesis techniques to ease the bus integration process. The first step in interface synthesis is detailed bus specification capture. Along with a processor model and its associated tool chain, this is part of a "Processor Support Platform" (PSP). During bus specification capture, the mapping of generic to specific signals and of transfer protocols to signal transition graphs are completed. This knowledge drives automatic synthesis of the logic needed to connect a processor to any peripheral, providing implementation of the user-specified communication scenarios. Since the PSP which is integrated into CoWare N2C, is provided for the designer in advance (usually by the IP/platform provider) the designer is shielded from needing to know detailed processor and bus interface. In effect, the detailed implementation level is abstracted to the Virtual Bus level. For the ARM processor and bus used in the pilot, the PSP is already captured in CoWare N2C.

3.2.3 Results of the Application

The result of this project is to be the assessment of the OCB and SLIF standards from a number of aspects of interoperability:

1. Interoperability of the VC. Following the SLIF documentation principles enhances all design groups' understanding of the VC so it can more easily be made to operate with other designs
2. Interoperability of the SLIF and OCB specifications. The pilot project shows the flow from the SLIF work to implementation via the OCB specification.
3. Interoperability of real buses with the VCI. Both manually, and automatically using Interface Synthesis in CoWare N2C, the pilot project shows that the VCIcan be implemented in practice with a real commercial bus - in this case for the ARM processor.

As we write, the pilot is still in progress, but early indications are that we will achieve our goals.

4 Conclusions

Standards at the system-level are helping to solve a real design problem: they are creating an engineering mind-set which encourages design for re-use. There are two key points to this: (1) the recognition of basic interoperability requirements, and (2) development of a common ground for mutual comprehension. As the VSIA is building around the concept of a “design-data” model and the clean separation of behaviour from interface, these system standards apply independent of adopted syntax and design environment. Besides the improved ease of integration, this standardisation approach ensures straight-forward cross-checking of the VC specification with the actual implementation.

Taking such a strong role in system-level standardisation places the VSIA is in the forefront of System-on-Chip design practice. This organisation has industry leaders both contributing to, and adopting our work. We remain separate from the development of system-level languages. However, our design-description techniques and interoperability requirements will play a crucial role in guiding the development of any new system-level language. In particular, the work of the VSIA helps to identify and prioritise the support of possible design-format features. Through this role as a language-neutral party we are defining authoring styles and constraint sets which will ensure the future interoperability of system-level virtual components.

5 References

[1] VSIA, *VSI Alliance Architecture Document v1.0,* www.vsi.org, March 1997

[2] VSIA, *VSIA System Level Design Model Taxonomy Document*, www.vsi.org, Jan 1999

[3] R. Goering, *VSI spec establishes system-modeling taxonomy*, EE Times, Feb 1999

[4] J. Rawson et al., *Interface Based Design,* Proc. of Design Automation Conf., June 1997

[5] C. Lennard, *Enabling VC Exchange through System-Level VC Standards,* Proc. of Forum on Design Languages, pp. 641-650, Sept 1999

[6] K.Suzuki, et al, *OwL: An Interface Description Language for IP Reuse,* in Proc. of Custom Integrated Circuits Conference, pp.403-406, May 1999.

[7] Ptolemy II *Heterogeneous Concurrent Modeling and Design in Java v0.1.1,* ERL Technical Report, University of California Berkeley, Feb 1999

[8] IEEE Std 1164-1993 *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*

[9] IEEE Std 1076.3-1997 *IEEE Standard VHDL Synthesis Packages*