Design and Use of a System-Level Specification and Verification Methodology

M. M. Kamal Hashmi & Alistair C. Bruce

Design Automation Centre, High Performance Systems,
ICL, Wenlock Way, Manchester, M12 5DR, U.K.

Abstract

This paper describes the problem of Design Capture at System level and of moving a design verifiably down levels of abstraction. We describe our steps on the way to designing a methodology which captures system level interface and functional specifications, and enables the designers to decompose and refine specifications down to RTL VHDL in a hierarchic and piece-wise manner.

1. Introduction

Design capture and verification at the earliest possible stage increases the commercial viability of a product by reducing the overall time to market.

One way to compress and effectively manage the complete product timescale is to formally capture the design earlier in the design cycle so that it can be verified earlier, and problems conforming to the requirement specification solved before reaching the later phases of the design cycle.

Ensuring the correctness of a design early on, has a beneficial "ripple" effect on later stages. This is magnified by the growing complexity of the design.

1.1 Traditional Design

Traditionally, in the electronics industry, system specifications and requirements have been captured as natural language documents. These specifications are then refined and elaborated in a series of manual design steps until they are detailed enough to be captured formally in schematic or textual form for simulation. Each such step is usually verified against the previous step – its specification – manually by review.

The first simulatable level that is formally captured is usually at Register Transfer Level (RTL) or below – though in the last few years there has been some experimentation with algorithmic capture in the CAD tools industry. Once captured, the design can be verified by simulation with test patterns or a test harness. It is then refined until it can be synthesised or manually translated into a gate level description.

1.2 Previous Design Experience

The Corporate Systems Division (CSD) of ICL has been involved in the design of large complex systems like mainframes for over 35 years. In the early days, the machines were captured at gate level after manual design. Later, the level of capture rose to library cell level which was equivalent to RTL

The design of the SX mainframe [1] was initially captured at an algorithmic level which was refined into RTL behaviours and then implemented at gate level. The use of extensive simulation at all levels led to a large reduction of faults in the first prototype compared to the previous mainframe despite being a more complex machine. Our simulation requirements meant that our in-house simulation capability was provided at high levels and high performance [2].

However, the next generation of very large servers, SY, were again going to involve another leap in complexity and total gate count. This, combined with a reduction in timescale, meant the number of design faults and redesigns had to be reduced even further at every level.

2. System Specification Problems

After the SX mainframe design was complete, an analysis of the problems that occurred during its design revealed three major causes of avoidable errors. These were:

a) The system went, in a single step, from a natural language specification to RTL. This produces more errors than going to RTL in a series of well understood, and formally captured, steps.
   Also, slight ambiguities in the specification meant that different designers wrote RTL models that did not always work together. Finding these differences of interpretation at RTL was time consuming, and the problems were often difficult to fix.

b) The system was broken down early into its major components because of its size and complexity. The interfaces between the components were ambiguously or incompletely specified leading to subtle interface mismatches when the whole design was eventually simulated.
These mismatches were *not* in the static connections, but in the actual behaviour of the units across the interfaces.

c) The highest, and hence smallest, simulation model of the entire system was at RTL.
This model was so large that its performance severely limited the amount of full system testing that could be performed – only a few million beats as against the requirement of hundreds of millions of beats.

2.1 Initial Targets

To solve these problems we needed a language or tool which would allow us to formally capture the specification at a high level unambiguously.

This tool should also enforce the separation of the specification of functionality and interface. A common independent interface specification could then be written to be used by the functional units at either end. This specification must not just define the connections – it should also specify the protocol of communication across the interface.

These specifications must be able to be captured at different levels of design, from an asynchronous level with vague or fuzzy timings to synchronous clocked RTL with precisely timed functional units.

Also important is the need for the lower level designs to flow naturally from their higher level specification such that, in a simulation, any high level specification can be replaced by its lower level design. This form of *Mixed Multi-level Modelling* would mean that each low level design could be tested in a system simulation without a severe performance overhead.

An important management requirement is that the different parts of the system can be designed at different rates by different teams working independently off common interface specifications.

So, in summary, what we required from a specification language or tool was:

- Separate Interface and Functional Specification.
- Multi-level Specification.
- Hierarchic decomposition of specifications.
- Mixed Multi-level Modelling.

We first investigated VHDL to see if it could meet our requirements.

3. System Design with VHDL

VHDL is good for the capture of low-level RTL and gate level designs, but is not a natural language for the actual *design* of systems.

From an engineer's perspective, the problems with VHDL for high level design fall into three categories.

3.1 Syntactic Overhead

Every language chooses a balance between insisting that a designer specifies *exactly* what is meant, and filling in any gaps, intentional or otherwise, that have been left. In VHDL, the overriding need was for the precise expression of specification and implementation.

However, at higher levels the need is for quickly trying out ideas and speedy design changes. Only when a design meets all the functional and performance requirements will it be frozen and specified in detail for implementation at the next level down.

The overhead of constant revisions to a VHDL model is onerous, despite recent changes to the VHDL standard [3]. Hence the plethora of tools appearing on the market which try to hide the VHDL framework.

So the language requirement changes at different stages of the design. Either different languages or a language tool which behaves differently at each level is needed.

3.2 Interface Level Changes

A system designer does not define every connection and function. Some parts of the design are left to the lower level designers. For example it may be specified that parity checks exist, but not the where and how.

So, it is essential to be able to add to, or change, a design's interface as you go down levels of design.

In VHDL, changing an interface entity is not a trivial task and usually the functionality will need rewriting to cope with any changes.

One of our aims is to co-simulate designs from different levels in order to reuse test harnesses and also improve test coverage and performance. This is not possible in VHDL if the interface has changed.

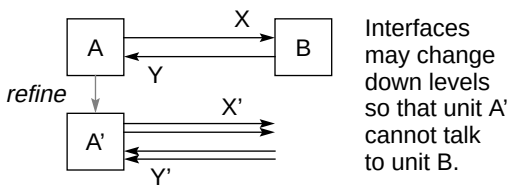

Figure 1: Interface Changes.

Additionally in a lower level design the timing of the interface protocols may be a subset of the higher level protocols and so, even if the interface is statically the same, its dynamic behaviour may vary.

In these cases, the designer would need to write, and maintain, a piece of VHDL to sit around one of the models and translate one interface into the other.

3.3 Design Paradigms

Design at a system level is often in terms of general paradigms rather than specific implementations. A system would be designed as units which can do certain things and which transfer the necessary information. The details of how a unit performs its functions and its static connections to other units are considered later.

VHDL is deficient at providing system level paradigms in at least three of the ways we require for a system specification and design language:

Firstly, the properties of VHDL processes and signals lead to a structural form of design whereas system designers tend to think of a system as a set of asynchronous communicating processes. In the design process, each of these and their communication protocols is refined and elaborated until a stage is reached where each unit can be considered as a set of communicating sequential processes. These are then implemented or synthesized into gates.

Secondly, a model in VHDL is of a fixed size with a known number of instances. This is obviously a requirement of any model at gate or RT synthesizable levels, but at system levels it would be very useful to have creation and termination of processes as needed by the simulation. This would not only make system modelling easier but would aid the evaluation of concurrency versus performance.

Finally, at system level, the designer would like to treat time as an elastic property. VHDL allows only absolute, precisely defined time steps, whereas a system designer would like to build in terms of concepts or paradigms like ‘sometime after’ and ‘between X and Y beats after’, and then simulate and analyse the resulting problems or the performance.

4. Initial work – the CHISLE Prototype

After deciding that VHDL would not solve the system specification problem, work was started on a methodology that would solve this problem for SY.

4.1 Interface specification

A language was developed to capture interface specifications which could be used to capture the interface between two units at multiple levels of abstraction:

- The *Transaction* layer defined the allowable conversations between units in terms of *Items*.
- The *Item* layer defined an ordered sequence of *Packets* from one unit to the other.
- The *Packet* layer defined an ordered and timed sequence of *Slices* from one unit to the other.
- The *Slice* and *Wire* levels defined the static connections between the units, and their direction.

Interfaces could also handle concurrency which was modelled as an array of the concurrent entity.

Time relations were specified between two instances as ‘after-end’, ‘after-start’ or ‘none’ with bounded or unbounded ranges.

4.2 Functional Specification

This language was then extended and a new dialect designed which could be used to capture the functional units that communicate using the interfaces. These were captured at three basic levels:

a) *Group* level units which used the *Item* layer of its interfaces to communicate with other units. Thus, as in items, a group level unit specified functionality using unbounded time relations.

b) *Set* level units used the *Packet* layer of their interfaces to communicate. So a set level unit was specified using bounded time relationships.

c) *Element* level units used the *Slice* layer of their interfaces. So, since slices are timeless structures of data, the element level unit specifies completely how it communicates via structured data paths with other units. They use interfaces just for static connectivity and for checking that their communication adheres to the higher level protocols. This level can be written at RTL.

4.3 The CHISLE compiler

The whole language comprising the two dialects was called CHISLE - Combined Hardware and Interface Specification Language for Engineers [4]. The language also contained normal language features such as type definitions and abstract data types like queues.

A prototype compiler was written which generated models for the in-house MSIM simulator. The compiler treated different levels of CHISLE in different ways – for example, the use of a particular timing construct which is allowed at group level may produce a warning at set level and an error at element level. Even the generated code differed depending on the level of the unit.

The interfaces allow a unit to communicate with other units at any level. A *group* level unit could send an item to a *set* level unit. Its side of the interface would break down

the *item*, generate the correct sequence of *slices* and send them on the *wires* to the other units. The *set* level unit would receive the *wire* data and its interface would check the data and build up the correct *slice* and *packet* before passing it on to the internal functionality.

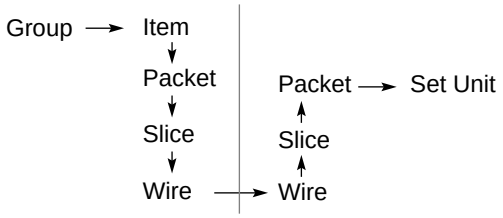

Figure 2: Group to Set Communication.

CHISLE is being used by the SY project and so far, the project has run over 100 interfaces and 400 functional units (including test specifications) through the CHISLE compiler. It has enabled greater testing and better model performance, and increased design productivity by a factor 3 to 4 times compared to the previous design.

5. Post-CHISLE Analysis

A review of CHISLE's capabilities and an analysis of the problems the engineers had with it indicated that there were some changes needed to improve the tool.

5.1 Design Clarity

When analysing how the engineers designed in CHISLE we discovered that much of the design still occurred off-line before they captured it in CHISLE. They could not capture some of their ideas in the most natural view.

The interfaces were usually designed using diagrams and then captured in the interface dialect. The functional dialect also did not facilitate a clear implementation of a unit's interface requirements.

1. Interface and functionality structure should be captured graphically to provide a clearer view of the design, and of how the units satisfy the interfaces.

The many fixed levels provided in CHISLE did not always lend themselves to all kinds of design. Designers had to fill in all the levels even though they only wished to use a couple for some designs.

2. There should be *flexible* multiple levels so that the system designer could choose the levels to use.

5.2 Language Reuse

When the requirements for CHISLE were generated the synthesis tools available were not advanced enough to meet high-performance engineering standards but they have improved enough that their use in parts of the design would improve productivity.

Most synthesis tools take VHDL input, and the emergence of VHDL as a standard HDL meant that we would use VHDL somewhere in our next design route.

3. The tool should be *based on* VHDL, such that the design would naturally flow into pure VHDL at RTL.

5.3 Even Higher Levels

Finally, as CHISLE was being implemented it became apparent that transactions were a powerful idea that had not been fully utilised. A more generic definition of a transaction could be used in functional units as well as interfaces and would enable an even higher level of functionality. It would then be possible to specify and simulate the complete dataflow of a system before specifying the control or data transformations – the dataflow performance of the system could then be analysed.

Extending transactions and other interface entities into functionality would allow the functional activities to have a dynamic temporal interface instead of just a static one. Designers could then grow internal interface components which could become real interfaces when the design was hierarchically decomposed.

4. Interface concepts should be extended and allowed in functional units.

The CHISLE timing and concurrency paradigms, while considered extremely useful, were not generic enough or simple to use. In particular the timing paradigm, when used heavily, made the design intent obscure rather than clearer.

5. The timing relationships should have a graphical view.
6. The concurrency paradigm should be extended to cover the different requirements at different levels.

5.4 Performance

The performance of the CHISLE system models was much better than the previous design's system model but a performance analysis showed that we could get another gain in performance of at least 3x by redesigning the models produced for MSIM.

At the time that the CHISLE compiler was written, the MSIM simulator worked only on VME mainframes and used S3 as its main behavioural language but it has been recently enhanced to run on Unix and take VHDL behaviours [5] – partly as a response to our post-CHISLE system specification review – and is also much faster.

7. By using the new MSIM simulator and the new model design, we expect to make the new System Specification Tool at least 5x faster than CHISLE.

6. The New Methodology

So, the extra aims we had for the new tool included:

- Multiple views and capture of the specification – much of it graphical – enabling the designers to captures his ideas in the most natural form.
- Paradigms for concurrency and timing which are easier to use at high levels and transform easily to lower level versions of the paradigms.
- Better Simulation Performance.
- Smooth transition to the VHDL RTL tool of choice.

6.1 Overall Structure

For SY, the average size of an interface was about 200 statements and of a unit was 1500 statements. We want as much of this as possible to have graphical capture with multiple views. This would make the design more comprehensible and the tool more designer friendly.

The idea is that engineers will not perceive that they use any textual language other than VHDL, the rest of the information being captured graphically – the tool would be a *supra-language of mixed graphics and text.*

The unit of design is an *interactive* document called a *Specification Document*. This can be one of the following types:

- *Interface Specification Documents* – for the capture of interfaces. Most of the information in an interface can be captured graphically with a little text.
- *Unit Specification Documents* – for the capture of functionality and associated data. We make functionality clearer by giving graphical views to the overall structure and the concurrency flow, with VHDL text for most of the rest.
- *Library Documents* – a necessary enabler for reuse. VHDL packages are an example of a library document.

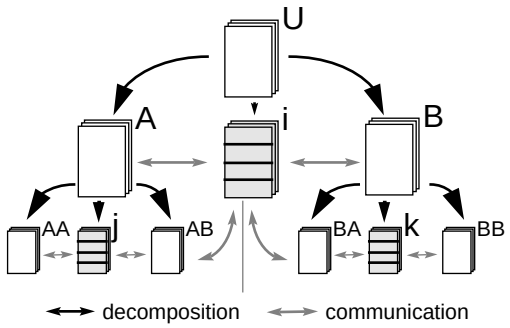

Figure 3: Specification Structure.

Unit specification documents reference the *interface specification* documents that they use, and the Units into which they decompose. Note that interfaces are in themselves multilevel – so in figure 3, unit AA would communicate with units B, or BA and BB, via the interface 'i'

6.2 Interface Specification Documents

Interfaces now have only two explicit levels. The lowest level, at which communication with pure VHDL models takes place, is composed of *signals*. The higher levels are composed of *packets* and *transactions*, which can now be hierarchically defined at any level.

The higher levels can be specialised by the attributes *loose*, *timed* and *timeless* which can be used to specify intentions about a level but they are not compulsory. So, for example, if an entity is *timeless* then it not allowed to have timing constructs in its definition or in its children.

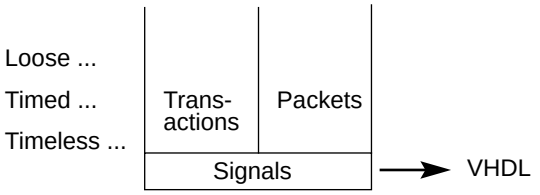

Figure 4: Levels of Interface.

Interfaces can now have one, two or more ends. Also, they no longer know to which units they are connected, so that *interface specifications* can now be reused between different units.

There are four main parts to an interface definition – all of which are optional:

a) The *services* definition specifies the throughput or capacity of the interface by defining which 'conversations' are allowed concurrently.

b) *Transaction* definitions still specify conversations across interfaces. Unlike in CHISLE, transactions can now define communication at any level from a loosely timed non-deterministic system level to a tightly timed RT level.

c) *Packets* define a one-way stream of information from one end of the interface to another end. They can be decomposed into other *packets* or into *signals* and can be defined at any level.

d) *Signals* specify static connectivity. If the designer wishes to co-simulate with VHDL models or to decompose into pure VHDL, this level is essential.

Communication via *packets* defaults to *asynchronous communication* with a channel size of one, however the tool can also model *synchronous communication* with a *rendezvous* [6].

6.3 Unit Specification Documents

This document captures three aspects of the unit:

- The *Functionality* of the unit.
- Its de-*Composition* into lower level interfaces and units. This is captured graphically, and can generate initial skeletons for the lower level documents.
- *Test Specifications* for the unit which are held as appendices to the document. These are used to test the unit and its decompositions.

The *Functionality* of the unit can contain *transactions*, *packets* and *signals* like interfaces. However, in *Interfaces* total communication always takes absolute time, whereas in *Units* communication can also take place in delta time and instantaneously.

The *services* definition specifies the transactions from the unit's interfaces that the *Functionality* satisfies.

Activities are the part of a *Functionality* which actually guide the running of the unit through time. Activities are mapped to transactions, and are expected to satisfy one end of the transaction.

Internally, an *activity* is composed of 'leaves' of VHDL code, and dependencies between these leaves which define when the leaves get activated. Using this mechanism it is possible to define arbitrarily complex braids of action, mixing concurrent and sequential leaves. The internal structure of an activity can be viewed and captured graphically.

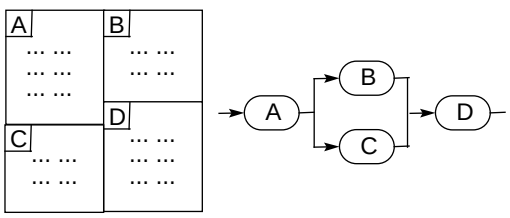

Figure 5: Graphical Activity structure.

We have also defined three kinds of concurrency:

- *Parallel*, when each activation is independent of the others and takes up resource.
- *Pipelined*, when each activation must be at a different state to the others and reuses resource.
- *Concurrent* which is either of the other two.

The user is allowed to specify a maximum number of concurrencies or ask for 'unlimited' where the tool spawns another whenever needed – this is useful at the higher levels when performance is still being investigated.

Also at the higher levels, the tool allows some non-determinism where a set of paths are tagged by booleans, and a fair choice is made of one of the paths whose tag evaluates to true.

7. Current Status

Design is complete and work on the production of the tools, using an extended version of MSIM as the core simulator, has begun.

Our target is to create a new product and methodology which will assist the System Designer, improve productivity and lead to better design methods.

Our primary objectives are ease of use and raw performance.

8. Future Possibilities

Since the supra-language formally captures all the design stages from a very high level down to a low level register-transfer level it becomes possible to investigate methods of high level synthesis.

Also the formal verification of designs at levels higher than netlists of combinational logic becomes available for investigation.

9. Acknowledgements

Grateful thanks go to Tony Jebson and Chris Jones who decided to go and actually do something with the results of the SX post-mortem analysis. And to the rest of the ICL hardware engineers who aided and abetted the development and validation of CHISLE; and Martyn Edwards and Joe Murphy of UMIST who kept a critical eye on the post-CHISLE re-design.

10. References

[1] G.P.Abraham, D.C. Freeth and H.Vosper: *SX Design Processes*, ICL Technical Journal Vol. 7 No. 2, 1990

[2] S.Hodgson: *A Multi-Level, Mixed State Simulator for Hierarchical Design Verification*, IEEE European Design Automation Conference 1984.

[3] *IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1993*, The Institute of Electrical and Electronic Engineers, New York, USA, 1994.

[4] A.Jebson, C.Jones and H.Vosper: *CHISLE: An Engineer's tool for hardware system design*, ICL Technical Journal Vol. 8 No. 3 May 1993.

[5] S.Hodgson, Z.Shaar and A.Smith: *A High Performance VHDL Simulator for Large Systems Design*, IEEE European Design Automation Conference 1995.

[6] C.A.R.Hoare: *Communicating Sequential Processes*, Prentice-Hall International, 1984.