Application of High Level Interface-based Design to Telecommunications System Hardware

Dyson Wilkes
Ericsson Components Ltd., UK
Dyson.Wilkes@swindon.ericsson.se

M.M. Kamal Hashmi
International Computers Ltd., UK.
Kamal.Hashmi@icl.com

Abstract

The assumption in moving system modelling to higher levels is that this improves the design process by allowing exploration of the architecture, providing an unambiguous specification and catching system errors early. We used the interface-based high level abstractions of VHDL+ in a real design, and in parallel with the actual project to investigate the validity of these claims.

1. Introduction

The design of complex electronic systems is a task that takes large teams of engineers a considerable time to complete correctly. As the design gets larger and more complex, the resource needed seems to increase out of proportion to the task. Comprehension of the whole design, communication of design intent between the different teams and, in particular, validation of the design itself gets harder and takes more time.

The traditional way to solve these problems is to describe the design at higher levels of abstraction so that larger parts of the design can be understood, captured and verified quicker. It is recommended is to consider orthogonal abstractions separately, and to capture this higher level unambiguously in a formal language. A good rationale for the methodology is given in [8]. In the last few years, the separation of the Communication from the Functionality in a design has been considerably investigated resulting in many favourable papers, including [4] and [6].

Initially based on an internal language [1], ICL has been developing an interface-based methodology with abstractions in time, process communication and resource scheduling, synchronisation and dependencies [7]. This methodology is being implemented in an extended pure superset of VHDL called VHDL+ [9], mainly to enable easy and seamless refinement of the hardware to a synthesizable Register Transfer (RT) level with the software parts implemented in C++ and communication via VHDL+ interfaces.

Ericsson has been using VHDL [2] in high level, hierarchical design flows for a number of years. While the benefits of creating models above RT level were recognised, the amount of effort involved in maintaining multi-level models had often proven too much to be carried on throughout a project. These models represented the design at different degrees of detail which are called abstraction levels. One significant problem arose when dealing with interfaces between the blocks in the design at different abstraction levels. The ability to simulate one block at a detailed level while keeping others at a more abstract level gave advantages such as faster simulation speed and less work required on testbenches. The problem with using standard VHDL was that it took just as much time to write and check the translators needed to convert the signals between the abstraction levels. Tools based on the VHDL+ language promised to solve this problem and improve other aspects of the design process for Ericsson's telecommunication products.

2. Project Overview

The SYSTEL project to evaluate the VHDL+ methodology set out to construct and use a multi-level system model of a control system for a large public switch called APZ-CP and to try to demonstrate improvements in the overall design process. The actual goals were the project were:

- Show if SuperVISE, the ICL VHDL+ tool, could be applied in the design of a telecommunication system. This had not yet been shown despite its success [4] in ICL for mainframes.
- Create an executable working system model of an APZ-CP in VHDL+.
- Refine one part of the design down to gate level to validate this part of the flow.
- Show if the design process was improved.

It was important to show both system level modelling and a clean path to implementation. The results of the project could be compared with a real production project of the APZ-CP that was going on at the same time. This was the best way to get an as objective as possible view on the degree of design process improvement. Even so, the SYSTEL project could not model an exact APZ-CP nor implement a large part of the system, thus comparisons had to be made with some care.

The design process improvement was to be measured in terms of a reduced time to market with equal or better product quality. Any increased opportunities for design re-use were also to be considered as valuable improvement.

3. The VHDL+ Extensions

VHDL+ is a fully compatible super-set of standard VHDL which is converted to simulatable VHDL by the SuperVISE translator. It has been presented publicly and is being proposed as an open standard via the IEEE DASC Systems and Interface-based Design Study Group. This is important since using a proprietary language considerably reduces opportunities for reuse.

The main extensions in VHDL+ add the ability to describe interfaces by adding the primary design unit *interface*. An *interface* is a collection of *messages* that may be sent and received by directional pairs of ends of that interface. The VHDL *entity* is extended to allow instances to be connected by *interface*s. Such a connection is known as an *interface instance* but can be thought of

as a channel capable of carrying the messages defined in that interface. The VHDL *architecture* is extended to allow *messages* to be passed over an interface instance with the *send* and *receive* statements. VHDL+ also adds a new way to describe and encapsulate system behaviour called an *activity*. These allow sequential and parallel behaviour to be more easily described than in VHDL. *Interfaces* encapsulate data types with how they are communicated while *activities* encapsulate data items with the functions that operate on them. The sub-sections give examples of these two areas of extensions.

Another important extension is the ability to increase the granularity of time from a precise time, e.g. in nano-seconds, to clocks and to indeterminate ranges of time and clocks. This can be used in activities and in interfaces.

3.1 Interfaces

An example of a simple interface with one top level message is given below:

```
INTERFACE csi IS
    BETWEEN client, server ;
    CLOCK clk EVERY 10 ns ; -- 100 MHz
    MESSAGE cmd_m( cmd : q_cmd_t ) IS
         FROM client TO server ;
         TAKES 1 clk ;
    COMPOSITION
    PARALLEL
         strobe('1') ;
         bit0(conv(cmd(0))) ;
         bit1(conv(cmd(1))) ;
    END MESSAGE cmd_m ;
-- defns of strobe,bit1,etc would go here
END INTERFACE ;
```

The interface *csi* is defined as having two ends *client* and *server*. A clock called *clk* is defined to be active every 10 ns. The message called *cmd_m* has one parameter of type **q_cmd_t** and is composed of three lower level messages: *strobe*, *bit0* and *bit1*. A function called *conv* converts the data in the composition. When a *cmd_m* message is sent, it results in all three low level messages being sent in parallel. As messages are a fixed resource, if there were only one message called *bit* in place of *bit0* and *bit1*, the duration of *cmd_m* would be extended while each call to *bit* completed. So, if *bit* lasted 1 clk the timing constraint of *cmd_m* would be violated. This would result in *assert* messages at simulation run time. The *takes* statement may be a range of time or clock ticks and can be useful in a specification model to give flexibility and controlled non-determinism or in a testbench to stress a design over its operating region.

The ranges are useful during specification whereas the clock-oriented timing makes the relation to RT level implementation very easy. It is also possible to apportion clock cycle time between the processing and communication of data by including VHDL+ *pause* statements in the unit architectures.

The interface description is of a declarative form and must be reversible. It has to take this form as a given interface might have either end at a lower level, requiring decomposition of a higher level message from low level messages as well as the reverse process of composition. Figure 1 shows how message composition works in a model. On the left hand side of the figure unit A is shown sending a high level message *mes*, while unit B receives its components; *aaa*, ... at a lower level. In this case the SuperVISE implementation of the interface description decomposes the high level messages into the appropriate collection of lower level messages. On the right hand side it is unit A that is working at the lower level, sending messages that are recognised as the components of a higher level message in the interface description. The interface composes these into the high level message which unit B is receiving.

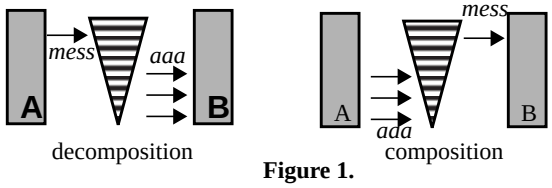

Figure 1.

This reversibility allows models to be built with components at different levels of abstraction. This is very useful because test benches can be reused at any level, components in a system simulation can be replaced by a lower level implementation and checked, and the design of different parts of the system can be done at different rates.

3.2 Activities

An example of activity usage for a FIFO used in the design is given below:

```
ACTIVITY fifo ( cmd  : IN q_cmd_t ;
                data  : INOUT q_data_t ;
                done  : OUT boolean ) IS
...
client : PROCESS
...
    fifo(put,d,stat); -- generate request

server1 : PROCESS
...
    fifo(get,d,stat); -- service request

server2 : PROCESS
...
    fifo(get,d,stat); -- service request
```

Activity syntax is similar to a procedure but an activity represents a fixed resource, like messages, and may be called with a maximum concurrency of one. If an activity is called from more than one place at a time, one caller gains access and the other has to wait. All callers will get access but the order of access is not defined. Also, an activity encapsulates data with the processing of that data because, like a process, it retains its variable values between activations.

These two properties are exploited in the FIFO example where the queue data and pointers are owned by the FIFO, thus the activity call mechanism provides the arbitration for simultaneous access. The code shows a client and two servers. Code has been left out for clarity. If we assume the client is able to generate requests faster than a single server there are a number of scenarios to consider, including: simultaneous access by a server and the client; simultaneous access by both servers. An example of these scenarios are shown in figure 2 where: the caller with access to the FIFO is shown as; c for client, 1 and 2 for the servers on the FIFO time line; a process waiting for access is shown by a narrow striped box and a busy server is shown by a wide grey box.

It is possible to observe the utilisation of the servers in the simulation to determine if the queuing scheme has the required properties and performance. In this example, the length of time

taken to put and get items on the FIFO is a critical factor. There is also a potential problem if the client cannot wait to be served, as might be the case if it had to be ready to receive messages from another unit or process.

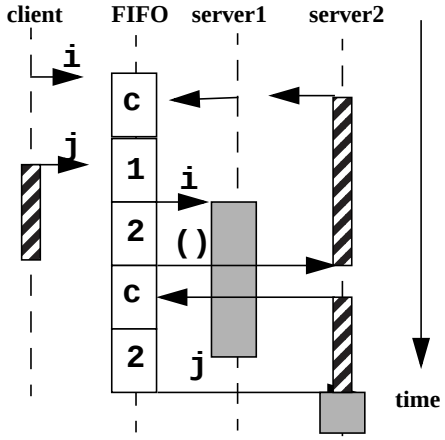

Figure 2.

Although this example is quite simple, it shows that a lot of complexity can be represented with just a few lines of VHDL+. The built-in message and activity mechanisms facilitate the use of a higher level representation of the interaction between units in a design. This gives VHDL+ the characteristics of a number of existing system level languages [3] [10] [11] but with the benefits of being compatible with the standard hardware implementation language VHDL.

4. Modelling the APZ Central processor

Ericsson's public telecommunications switch AXE-10 consists of two main parts: the control system APZ and the switch fabric. The control system is distributed and has a central processor (CP) to co-ordinate overall control. This section gives an overview of the central processor as it was modelled in the project. The APZ-CP was constructed of a number of ASICs totalling about 2 million gates.

Table 1. gives a summary of the abstraction levels defined for the project. These abstraction levels, although of potential use in other control dominated designs, are not likely to be of general application. This would be true of any set of abstraction levels as they are a function of the design domain and the needs of the design process. There were four levels in all, going from the most abstract first "Block" level to the least abstract Register Transfer level. It should be noted that a set of abstractions such as these are intended to make the design tasks more tractable. It should not be assumed that the whole system has to be modelled at all levels of abstraction. In the project, it was assumed that the CP communicated directly with the switching parts of the AXE-10 system. Therefore a top level view of the system and its environment consisted of connections to the telephone network, called **switch** and the control system, called **CP**, as shown in figure 3.

The switch communicates with the CP using so-called **signals**. A **signal** from the **switch** might occur when there is an event in the network, for example, when a subscriber takes their phone off-hook. On receiving a signal the CP must carry out some operation and may need to send a **signal** back to the **switch**. This is the highest level (most abstract) view of the system as was only used for the testbench In the APZ-CP the processing is done by executing software **blocks** and the act of doing this is called a **job**. This, then, is the next level of detail and the highest level of abstraction used in the system model.

Level → *Granularity ↓*	**1:Block**	**2:Step**	**3:Instru-ction**	**4:RTL**
Interface	Signal	Step	Instr. Cycle	System Clock
Operation	Job	LCS and Jump	ASA Instruc-tion	Micro-instruc-tion
Data Item	Call, Sub-scriber	Signal data	Prog. register	bit vec-tor, bit
Function	Block, device	Finite Abstract Machine	Unit	Logic block

TABLE 1.

A software block consists of a set of instructions including flow control and memory access operations. As a specialised processor like the APZ-CP evolves and hardware becomes more compact, it follows that frequently used functions come to be implemented in hardware. Whilst it might be difficult to predict exactly which parts of the system function might be moved to hardware it is possible to introduce a level of abstraction which decouples the levels described so far from the instruction level. This decoupling increases the possibility of model reuse and allows exploration of different dimensions of the design space. Given that processor instruction sets are one or two steps below the level at which modern software is developed, it seems appropriate to introduce an additional abstraction level into the description of the system. When high level software code is compiled into machine instructions the code is grouped into small collections of sequentially related instructions called basic blocks. Taking this as a cue and adding the fact that functions that might be implemented in hardware are likely to be equivalent to small, repetitive code segments, the concept of a software step can be introduced. A block, then, consists of a set of steps and each step is a set of instructions that operate on a finite set of data. This set of Finite Abstract Machines (FAMs), is the next level of abstraction in the system description.

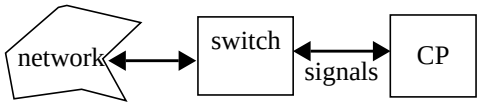

Figure 3.

5. The System Model

Figure 5 shows an overview of the System Model specified in VHDL+. This level of structural model was used to explore the system at both job and step levels of abstraction by creating different architectures for the **sched** and **proc** units. This model of the CP has two kinds of sub-unit, one called ***sched***: the scheduler and the other called ***proc***: the processor. The architecture is a multi-processor with separate and private programme and data memories. The first processor instance (**proc** 1) is specialized so that its data store acts as the central programme store for the other proc instances.

Working from the left of the diagram, the CPB (Central Processor Bus) carries messages containing APZ signals between the switch and the CP. These enter the scheduler unit which maps the signals into jobs which are enqueued awaiting the processor capable of running the job. When there is a capable processor, the job is sent over SP (scheduler-processor bus) via a *job_start* message which contains both the job and the target processor identity. The sched unit expects a proc unit to send a *job_end*, signal_m or *null* message. If there is no job for a processor it is sent a *poll* message to which sched expects a response. If there is no such response within a time-out period the proc instance is marked as faulty.

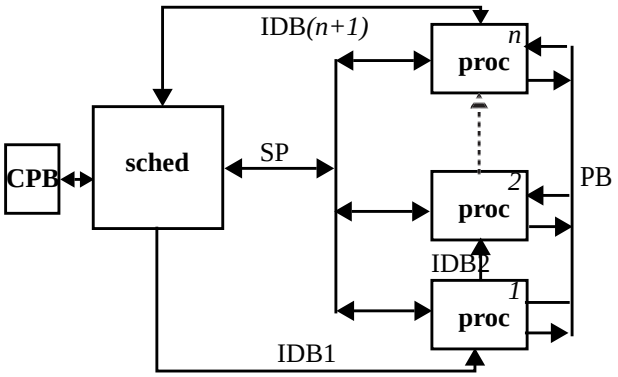

Figure 5.

The scheduler keeps track of the "capabilities" of each processor. A capability is the ability to run a software block and each kind of job requires a distinct block to be able to be run. If no processor has a particular capability then the scheduler selects the least loaded proc unit and sends a load_block job to the pstore proc. On receipt of the *job_start* the pstore proc sends the programme image to the appropriate proc via the programme bus (PB). Once a proc is loaded with a block it is the only proc that may run that kind of job so data consistency is guaranteed. The execution of a job consists of the execution of a set of steps. The sched unit directs the signals from the proc units and, where appropriate converts them to jobs adding them to the queue.

At system start-up the scheduler sends a message with parameter value equal to one on IDB1. Each **proc** unit in the daisy chain takes that value as its identity and adds one to the number before passing it to the next **proc**. The last **proc** is connected back to the **sched**. In this way the **sched** unit learns how many processors there are in the configuration. All of the **proc** units are then polled to check the function of the sched-proc interfaces. If a given **proc** unit does not respond before a time-out then it is marked as faulty.

6. Model Refinement

The **proc** and **sched** units were further refined to lower levels of abstraction. They were modelled with job and step level behaviour and then structurally decomposed. Finally, one of the components of the **sched** unit was refined down to RT level and synthesized to gates to prove the design flow.

The architecture of the **sched** unit consisted of separate control and data paths. However, both control and data messages were included in a single interface description because at this level the internal interfaces of **sched** were being specified and too many structural details would have over-constrained the implementation options. At level 1 the queue in **sched** was represented by an activity while at level 2 it became a separate unit called **sched_q**.

The design of the **sched** unit at level 2 consisted of devising an architecture and mapping the activity calls and message sending into commands over the **cdbus**. Figure 6 shows the structure of the **sched** unit.

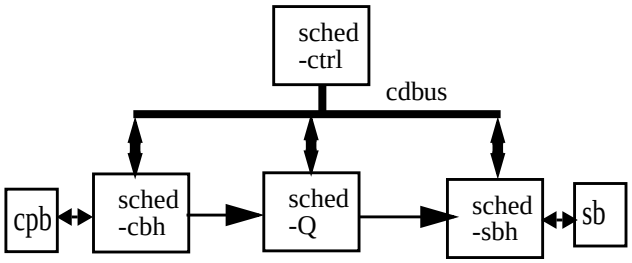

Figure 6.

At level 2 the processor's structure was as shown in figure 7. The processor unit was partitioned into four sub-units at this level. The processing core was called **proc_eng** and it requested enqueued jobs from **proc_q**. The programme and data memories were embodied in **proc_ps** and **proc_ds** each of which connect to the common external system programme/data bus.

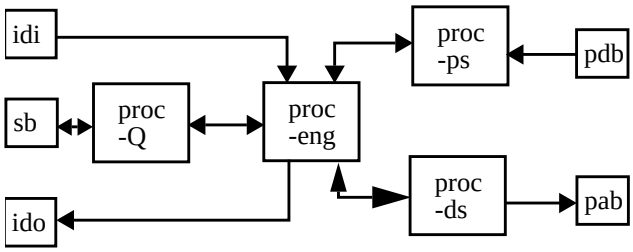

Figure 7.

The processor engine core within **proc_eng** was modelled down to abstraction level 3 to show the mapping from step to instruction level execution. At this level, three units **instru**, **alu** and **reg_file** formed the kernel of the machine. To retain as much flexibility as possible, the communication was still modelled using messages at this level. In this way it was simple to construct a pipelined processor using the send and receive VHDL+ statements to define the interaction between units. The interface-based approach also assisted in this partial decomposition process.

The interface between the sched unit and proc units (labelled SP in figure 5) was designed down to gate level to show that the methodology was viable. This work also provided data on the efficiency of the design process, both in terms of the effort needed and the quality of the result.

As a consequence of this, the **sched_sbh** sub-unit of sched was designed down to RT level, and thence synthesised to gates. The RT level implementation of sched_sbh consisted of a Finite State Machine (FSM) and three other units controlled by it: cntdwn5 (a 5 bit settable down counter), timer and polled. The FSM was mapped directly from the interface description with some optimization for speed and area while the timer and polled units related to the activities at level 2 and the counter was used for control of the variable length part of the messages.

The unit was synthesised to a 0.35 μm gate array library using the Synopsys Design Compiler version 9701. The constraints for synthesis were derived from the interface specification. The

resulting total number of gates in the design was 5200 with 4200 of these implementing the FSM. The design, synthesis and gate level verification of the sched_sbh module took three weeks to complete. As VHDL+ language implementation at packet level – the layer between messages and VHDL signals – was not complete before the end of the project, a simple work-around using wrappers was used to perform the translation between the RTL signals and the low level messages.

7. Design Process Benefits

Using VHDL+ in the design flow increased the amount of work that could be done at the higher levels and thus how much verification was done at these levels. This, in turn, reduced the number of bugs found in the later stages of design and eased implementation and integration.

7.1 Architectural Exploration

The main benefit of VHDL+ over VHDL is that complex interactions between objects in the design can be hidden in the interface descriptions and reused between interface instances. The activity call mechanism also increases the possibility for data encapsulation and hiding. Due to the ease of designing at this level, two variations on the top level architecture were explored in the project. One had a separate **pstore** unit rather than the specialised **proc** instance. The other was in an earlier version of the architecture which had all the queues in the sched unit. In both cases VHDL+ speeded up the design process compared to using standard VHDL. For example, the activity definition for the queue function was copied into the level 1 proc model and calls inserted in the existing code. The pstore functionality was added to the processor by moving the activity from the special unit to the proc unit code.

Using a modelling language also helped in the evaluation of the design alternatives. The interface descriptions with the message duration feature allowed the architectures to be evaluated against some feasible timing constraints. Once these had been determined, the values formed a specification for the next more detailed level of implementation It was easier to spot inconsistencies and errors in this form of description than in a traditional paper document.

7.2 Partitioning and Decomposition

The decomposition process involves the breaking down of hardware and software units at one level into smaller implementable units that interact to yield the required functionally. The partitioning and decomposition processes, therefore, interact with each other: units are decomposed and then functions are partitioned from one level down to a set of interacting lower level units.

VHDL+ helps in these processes by separating the concerns of describing the function of each unit from those of specifying the interaction between them. This assistance was evident in the project at almost all levels. For example, the details of the communication between **sched** and **proc** were kept in the **sp** interface description. Also, when the sched unit was decomposed to level 2, the communication between **sched_ctrl** and the other **sched** sub-units was mostly transparent.

7.3 Implementation

The scope of this project was limited to the implementation of the hardware aspects of the system. It is hoped that the model would be suitable for hardware-software co-development at some level but it is recognised that suitable links to software emulation systems would be needed to fully support detailed software development. The structured organisation of function and communication descriptions in VHDL+ helps in this phase of design also. The declarative nature of the interface description makes for a clearer specification of that aspect of the design compared with a typical behavioural model written in VHDL.

At each stage in the design process, VHDL+ helped speed up the task of capturing and verifying the design. Implementation details could be added to interfaces or functional parts without too much interference between each aspect. It was very easy to partially implement some aspect of the system, for example, the **proc** unit engine was implemented with only its core functions at level 3. This provided the opportunity to evaluate the micro-architecture in the context of the entire system.

The transition from suitable VHDL+ to synthesizable RT level code was relatively smooth except for the lack of the packet level which describes messages in terms of VHDL signal transitions. It was possible to map activities and message compositions into familiar RT level constructs such as FSMs or register and multiplexer combinations. The increased opportunity for design re-use at this level was shown in the design of the buffer and queue in the **proc_q** unit. With some additional work certain VHDL+ constructs might be identified as implying common structures like FIFOs, queues, stacks and circular buffers. The ability to imply such elements could raise the level of automatic design synthesis above what is currently available in commercial tools

7.4 Verification

The generated VHDL from SuperVISE was simulated on Mentor Graphic's QSIM 5.0 on a Sun UltraSPARC. The stimulus for the model was provided by a random poisson arrival process modulated by a two step Markov chain to model bursty traffic [12] [13]. Simulations were done on a number of configurations of the design:

- stb2_l2_cfg: All at level 2
- stb2_sbh_wrapper_cfg: **sched_sbh** at RL level
- stb2_l3_proc_cfg: **proc_eng_core** at level 3
- stb2_q_wrapper_cfg: **proc_q** at RT level.
- stb2_l2_rtl_cfg: **sched_sbh** and **proc_q** at RTL level.

All of the above configurations contained six instances of the proc unit. In addition level 2 configurations were created with 4, 8, 16 and 31 instances of the proc unit. These were used to test the scaling of the system capacity. The simulation of a 6 processor configuration running at level 2 ran at a rate of 10 jobs per second. Comparing this with the fastest simulation of the RTL model of a single real processor which was 300 cycles per second suggests that the abstract model ran 2000 times faster. This number is based on the fact that a typical job consumes 4000 cycles. This speed-up allowed a considerable amount of simulation to be done giving a lot of very useful performance and capacity information. These results and graphs are included in the public SYSTEL report [14]. The high level stimulus described above proved to be very effective in revealing design faults. Similarly, the self-checking nature of the interface models increased the bug coverage [15].

8. Results and Analysis

The design activity in the production project relating to the units which were comparable to those designed in this demonstration project was collated and analysed. The data was based on the number of lines changed in the RT level code in a given month.

To be able to compare the two design processes, it is necessary to scale the design effort according to the design complexity. From the synthesis results **sched_sbh** was implemented by approximately 4000 gates. This figure discounts the job buffer flip-flops as this was implemented as a memory in the production design. The comparable production circuit was approximately 10,000 gates, excluding memory blocks. This gives a scaling factor of 2.5. While the design work was done a total of 110 man-hours (approximately 1 man-month) of effort were recorded.

For comparison, the production design effort on the block equivalent to **sched_sbh** can be taken to be four man-months which works out to about 500 man-hours expended. This gives a ratio of 110:500 or 5.6 compared to the 2.5 from the gate count ratio. Since many project time estimates are based on a linear relationship between number of gates and time to design this comparison seems reasonable. It can be concluded that the design effort from VHDL+ to a synthesised and verified gate level implementation is a little less that half that observed in the production project. The detailed figures for the above calculation are given in [14].

8.1 Using VHDL+ as the Specification

Unlike a specification written in a natural language, the VHDL+ model (unit model and interface model) is complete and unambiguous. Its behaviour may be explored by simulation, and when an RTL implementation is written the functional equivalence with the VHDL+ model can be verified by simulation. Provided that essential VHDL+ concepts like message and activity are well understood and simple ways to represent them in synthesizable VHDL are found, using a VHDL+ specification makes it possible to produce correct RTL code in much shorter time than if a only a natural language specification were used.

One possible drawback of using and relying only on the VHDL+ description is that it does not clearly show the design intent and the reasons behind different design decisions as clearly as a natural language description might. In some cases a brief description supporting the interpretation of the code could be useful. There are a number of classes of system requirements which cannot be embodied in VHDL+. Examples include physical characteristics such as size, power consumption. All of these aspects would still remain in the textual portion of the specification. It would be appropriate to keep the document as the overview and use the VHDL+ model as an annex covering the aspects the VHDL+ handles well.

9. Conclusions and Future Work

We have used the SYSTEL project to demonstrat the use of SuperVISE and VHDL+ in the design of a telecommunications system. The system designed had some similarities with the mainframe computers developed by ICL using the methodology but the design methodology used in the project was very close to that used on, for example, ATM switching systems in Ericsson. Since the project was completed the transaction and packet levels were added to SuperVISE/VHDL+ making it an extremely powerful tool for systems level design of telecommunications systems.

It would be valuable to investigate how to integrate software development with the SuperVise methodology. For example, how acceptable would step level be to software developers? Can a compatible set of abstraction levels be found? Is it feasible to think in terms of a common language from which a combined hardware/software solution is created? Should this language originate from the hardware or software domain?

10. Acknowledgments

This investigation – the SYSTEL project [5] – was funded by the European Commission as ESPRIT project number 23909 under Directorate General III.

11. References

[1] 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.

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

[3] Anders Olsen, Over Færgemand et al.: *Systems Engineering Using SDL-92*, Elsevier, 1994.

[4] M.M. Kamal Hashmi and Alistair C. Bruce: *Design and Use of a System-Level Specification and Verification Methodology*, IEEE European Design Automation Conference 1995.

[5] Dyson Wilkes 1996, *SYSTEL Project Proposal*, EKA/NR/W-96:136. Ericsson internal document.

[6] J.A. Rowson and A. Sangiovanni-Vincentelli, *Interface-based Design*, Proceedings of the 34th Design Automation Conference 1997.

[7] S. Hodgsom and M.M.K. Hashmi, *SuperVISE - System Specification and Design methodology*, ICL Systems Journal Vol. 12 Issue 2 November 1997.

[8] A. Sangiovanni-Vincentelli, P.C. McGeer and A. Saldanha, *Verification of Electronic Systems*, Proceedings of the 33rd Design Automation Conference 1996.

[9] M.M.Kamal Hashmi, ICL: *VHDL+ Language Reference Manual*, Available on-line at http://www.icl.com/da.

[10] F. Belina, D. Hogrefe, A Sarma; *"SDL with Applications from Protocol Specification"* Prentice Hall, 1991 (*SDL, ITU Recommendation Z.100*).

[11] Kenneth J. Turner(editor); "*Using Formal Description Techniques - An Introduction to ESTELLE, LOTOS and SDL"*, Wiley, 1993 (*LOTOS, ISO/IEC 8807*).

[12] R.B.Cooper, *Introduction to Queuing Theory*, Edward Arnold Ltd., 1981

[13] J.F. Hayes, *Modelling and Analysis of Computer Communication Networks*, Plenum Press, New York, 1986

[14] *Project 23909: SYSTEL - Final Report*, European Commission

[15] Yossi Malka, Avi Ziv, Design Reliability - Estimation thourgh Statistical Analysis of Bug Discovery Data, Proc. DAC 1998, ACM