

Advanced Multiprocessor System Modeling

John Shackleton, Todd Steeves
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

shackleton_john@htc.honeywell.com

Abstract

Robust modeling is a critical ingredient for a successful multiprocessor system design methodology. This paper will present details of VHDL-based modeling developed by the Honeywell Technology Center. This modeling approach emphasizes multiprocessing and distributed communications with accurate and flexible workload representations. The resulting performance analysis can be applied to multiple levels of abstraction for both software and hardware.

In the process of designing and implementing our performance models, we have stretched VHDL from its traditional hardware origins to a great extent, discovering the advantages and the limitations to the language. This paper highlights both the advantages of VHDL for complex system modeling, as well as several maneuvers required to work around significant language limitations.

1 Introduction

As system architectures become more and more complex, early stages of the system design increasingly rely on tools and techniques that accentuate the most important design issues and uncover potential flaws. Concentrating on the early design stages has become a very important step in the effort to keep development costs down, and the flow of the design as robust and flexible as possible. The combination of performance modeling and VHDL has proven to be an effective environment through which the design needs of complex systems can be addressed [1][2]. The Honeywell Performance Modelling Library (PML), developed at Honeywell Technology Center, is a natural extension of this combination. The PML is a detailed collection of modeling constructs built upon VHDL, that provide the designer with a solid foundation for creating performance models. In addition, the PML is intended to facilitate rapid

prototyping, through its modular, interchangeable sub-components, and hardware/software codesign, by elevating software to a level of importance previously reserved for hardware. Such emphasis is still rare in the realm of VHDL-based design.

2 Hardware/Software Modeling

2.1 Processor Modeling

A critical factor in evaluating performance of complex system architecture is accurate characterization of the processor(s) contained in the design. Tools supporting the evaluation must have the capability to model software (if appropriate) with the full detail of actual code executing in the system. In addition, the driving requirement for many design methodologies is the ability to model the highly complex software and hardware level interactions. As a result, much effort was expended to develop a flexible, high fidelity processor model for the PML. This model provides a wide range of software modeling capabilities: from modeling of actual code to high level data flow representations. The software modeling is built on top of the full capability of VHDL. Additional features have been added to handle interrupt service, preemptive tasking, task communication, task synchronization and other services one would expect from an executive or general purpose operating system. The scheduling component can also support static and dynamic tasking and even rate monotonic scheduling.

The processor model can automatically provide reports that detail processor task activity timelines, missed deadline reports, processor utilization, task utilization profiles, and overall latency.

Figure 1 illustrates two types of processor models, the Light Weight Processor and the Multitask Processor models. The Light Weight Processor (LWP) model provides

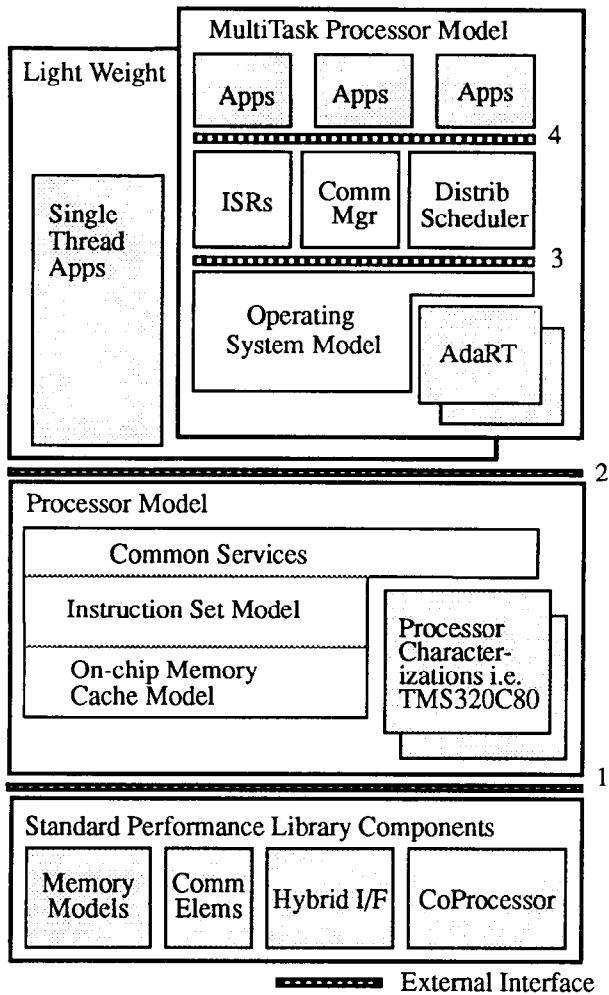


Figure 1: Processor Model Architecture

the equivalent of a bare processor. The LWP has less overhead and provides a more efficient software modeling platform for large multiprocessor simulations. All required system services such as communication and scheduling must be supported directly in the software application models.

The MultiTask Processor (MTP) model supports all the services one would expect of a commercial OS as well as a set of distributed multiprocessor OS services. Both forms of the processor model can be used together in a single simulation as shown in Figure 2. This brings the best of both to bear on large multiprocessor applications. The MTP model would be utilized for control and scheduling and the LWP would be used for dedicated applications where scheduling and multiple threads are not necessary.

The multitask processor model contains a well defined uni-processor operating system interface, denoted as interface 3 in Figure 1. The highest level interface, identified as

4, provides the distributed scheduling and communication services. This includes network addressing, mailbox, and name services capability.

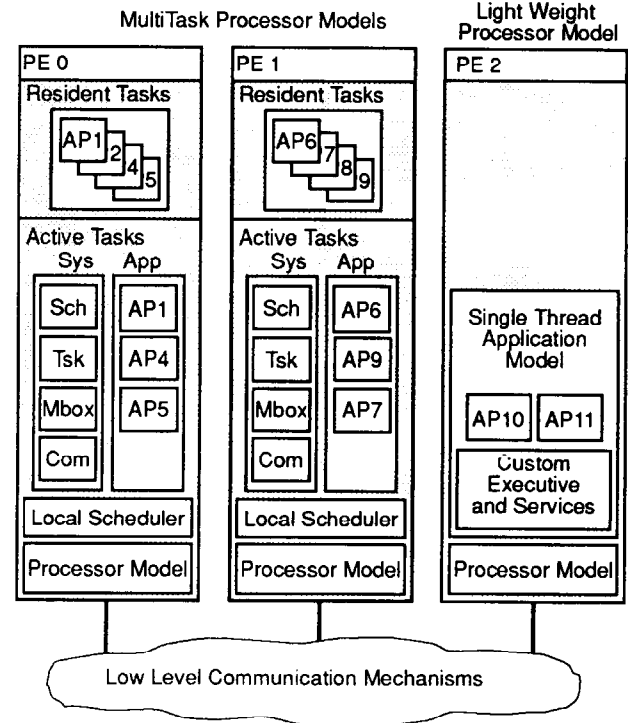


Figure 2: Multiprocessor System Simulation

Both processor types are supported by a single core processor model. This core processor model supports the interface, denoted as 2 in Figure 1, which can host single thread applications directly or provides the platform for supporting the complete suite of operating system services, multiprocessor communication, and distributed scheduling.

The processor model can interface with any of the wide variety of off-the-shelf performance library components via the standard performance token. Shown as interface 1 in Figure 1, the processor can interact with memories, communication components, and co-processors from the libraries. If necessary the user can supply custom components which utilize token application specific fields. The processor software components can read/write any of the token fields for transfer of application data to and from external components.

2.2 Software Modeling/Representation

In digital systems, software accounts for an increasingly large portion of the overall system functionality. Often preliminary software designs are never evaluated or simulated against candidate hardware designs. These two design activities are typically completed nearly indepen-

dently (the attitude being that the software will somehow “take care of itself”). The software and hardware designs often don’t meet until late in the integration stages. At these later stages, any problems encountered by mismatch of the two designs can have serious consequences on requirements, cost, and/or schedule. The PML, specifically the processor model, provides a means to simulate both hardware and software designs very early in the development process. At this development stage any problems uncovered can be remedied very cost effectively. The processor model provides an essential link between hardware and software that can be used from the very early design phases through detailed design.

The level of hardware model necessary to support software models is quite flexible and need only contain hardware performance level detail. As a minimum, a single processor model can be used with its internal memory/cache model. As multiple processors are added to the design, communication components such as network interfaces and cross-bars must be added and the system topology captured. If processor/memory bus contention between multiple bus elements is an issue, the processor memory model must be extended to reference external memory components.

Software models can be developed at various levels, from performance level to a completely detailed functional model. An important aspect of the processor model is its ability to support functional software models on performance level hardware models. If it is necessary for functional software components to interface with functional hardware components, special adaptors known as hybrid interfaces are required. Functional data exchange will occur outside the standard token interface in these cases. This is a topic of ongoing research at Honeywell Technology Center [3].

Often one begins with a performance level model of software which is essentially a high level data flow representations of the software architecture [4]. This preliminary model aids in identifying critical areas of the software that will possibly require more detailed model development. The user may decide to functionally model all elements of the software or just the critical areas. In either case, these functional models can be taken all the way down to individual instructions of the final software. The hardware architecture can be evaluated against the evolving software design.

One other possible scenario is when the designer begins with existing algorithms in pseudo-code or another programming language such as C or Ada. The first step in this case will be to translate those algorithms into VHDL, which is uncustomary since VHDL is not often thought of as a language for software or high-level modeling. Never-

theless, VHDL is a fully expressive language, so the translation of the algorithms is straight forward (more discussion on this point is made later in the section ‘Software Design and VHDL’).

Once translated, the designer has a vehicle for the subsequent detailed design activities. These algorithms can initially be hosted on a very elementary model of the hardware, as simple as a single processor, memory, and communication components. Only when these pieces are brought together can the hardware and software designs be performed collaboratively.

The main software design activities will be development of the architecture which include partitioning, allocation, scheduling, communication and possibly some hardware/software trade-offs. The hardware activities will include processor evaluation, processor clock rate, processor count, communication network bandwidth and topology.

2.3 Multiprocessor Communication

Extending the performance models to a parallel processing environment required extensions in the physical communication model which relate to the lowest levels of OSI stack, processor model extension to accommodate portions of the network layer extending up through layers 4 to 5, and distributed OS capabilities such as remote task execution. Figure 3 shows how the low communication layers do not provide adequate services for application processes to communicate in a multiprocessing environment.

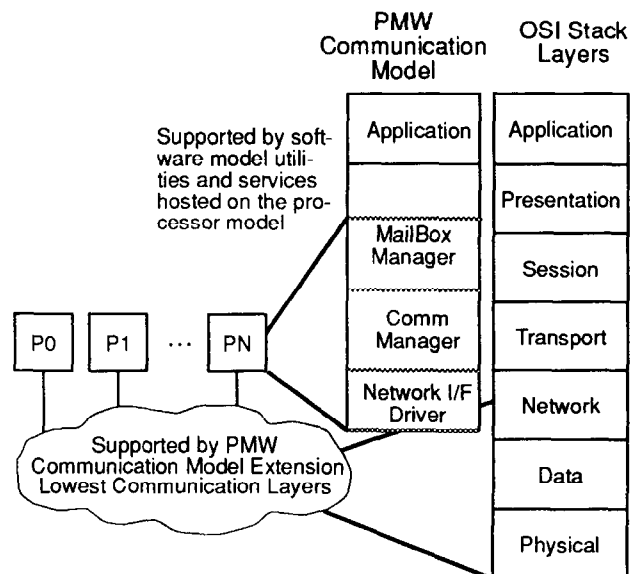


Figure 3: Communication Architecture

The communication layers are supported through the communication model extensions. This will include primarily the physical and data layer and a portion of the network

layer. Models will be enhanced to support generalized link types, access modes, and operational modes. Access modes include: frequency division multiplex, time division multiplex, and demand access with recovery. Operation modes are primarily circuit switched and packet switched.

Software models hosted on the processor will support services from the network layer and higher. The primary objective of this capability is to provide a process unique mailbox service which is accessible throughout the network. The upper most levels of the OSI such as presentation level will likely not be included in the models capabilities.

2.4 Communication Services

2.4.1 Addressing

One of the key elements of multiprocessor communication is network addressing. The address consists of a four tuple with the following fields shown in Table 1. Processor ele-

Address Form	Bits	Full Processor	LightWeight Processor
Processor Element	8	1..PE_MAX	1..PE_MAX
Process/Task Type	8	1..TT_MAX	Don't Care
Thread Number	4	1..TN_MAX	Don't Care
Port Number	4	1..PN_MAX	Don't Care

Table 1. Network Address Structure

ment is a physical processor number or identifier. Virtual to physical processor element translation is not necessary. Task type is a unique id for all application tasks available on the system. Two instances of the same task executing on the same processor will have the same first two field of address. They will be distinguished by the Thread Numbers. For dynamic tasks, a unique thread number will be assigned for each instance and will no longer be valid once the task has terminated. A unique thread id is supplied by the scheduler when the task is spawned. Rate monotonic tasks will utilize the same thread id for ongoing execution. Port number is an optional field that may be needed to support sophisticated communication structures which could require multiple ports per thread. Tasks may request additional ports which are accessible through the port number field of the address. These additional ports will be visible both to internal and external processes. Port number 0 will be the default port which is assigned at task instantiation. The LWP model communication layers do not interpret the task type, thread id number, and port number fields of the address but instead pass this information directly along to the single active thread. The task type, thread id, and even port id can be interpreted by the single application.

2.4.2 MailBoxes

Every rate monotonic and active dynamic task is assigned a unique event id which operates as a token message queue or mailbox. These are the same tokens utilized throughout the modeling environment for both hardware and software. The management of event id assignment is performed by the Mailbox Manager. The Mailbox Manager also handles the translation of network task addresses to application tasks, which involves the translation of both event id's and task slots. When messages or tokens are received by the Network Interface, the Mailbox Manager reads the address and directs the message to the appropriate task. During task initialization, particularly for Rate Monotonic Scheduling (RMS) tasks, the task will register its task type. Dynamics task are registered with the Mailbox Manager at the time they are spawned.

2.5 Operation

The basic network address operation and structure for a typical processor is shown in Figure 2. Each task thread will be assigned a unique mailbox for receiving data. The Mailbox Manager will receive all incoming network messages from the communication manager through a single mailbox. The messages will be sorted and distributed to the appropriate application mailboxes. Messages sent by all local application tasks will be sent to the Mailbox Manager's mailbox. If the mail is local, based on the PE id check, the address is looked up and the mail sent directly to the mailbox for the local task. If it is not local, the mail will be passed down to the communication manager to be sent out on the network.

Most of the local interprocess communication will not be supported on the LWP model. Since the LWP supports only a single thread, any network address with the appropriate PE id will received by the single application thread. The single thread application can interpret these addresses, including task type, thread id, and port number and then send out the appropriate responses. These response can be to other LWP in which case the task type and thread may or may not be need, but any communication to a MTP model will require these fields to be set.

2.6 Distributed OS Capabilities

The scope of activities required to develop a distributed operating system is large. In order to keep the problem well bounded, the PML provides a basic set of distributed services that are capable of supporting high-level scheduling models. Rather than define and implement a complete set of high level scheduling services we believe the resources should be applied to providing a general well designed basic set of distributed services as a foundation, which can be easily evolved into the needed capabilities.

PE Id 12: Active Tasks Mailbox Assignment

Mailbox Address	Task Type	Task Thread
Application		
12,8,0,0		8
12,8,0,1		8
12,9,0,0		9
12,9,0,1		
12,9,0,2		
12,9,0,3		
12,9,2,0		9
12,9,2,1		
12,9,2,2		
12,12,1,0		12
System		
12,0,1,0 Distributed Sched		0
12,1,1,0 Mailbox Manager		1
12,2,1,0 Comm Manager		2
	⋮	
12,7,1,0 TBD		7
Nonaddressable Components		
	Internal System Communication Mechanism	

Figure 4: Task Network Addressing

This will require developers of application models to embed some of the high level scheduling directly into their applications models. The distributed services that will be supported are centered around the capability to transparently, with respect to location, spawn, communicate with, control, and terminate tasks.

Figure 1 illustrates the major components utilized for multiprocessor communication and scheduling. In addition to the main scheduling component which previously has been integral to the processor model there now exists a number of system level components necessary to support the high-level services of a true multiprocessing environment. Included here are new scheduling and communication components which will handle most of the needed distributed services. The light weight processor model shown as PE 3 can be used interchangeably with the multi-task processor model. The software application model developer will be responsible for providing the appropriate functions for communication and scheduling. As a minimum, the application must provide the network

addresses in the correct form and observe some of the aspects of the network protocol not handled within the multiprocessor communication models. The current communication models roughly support OSI Layers 1-3. It may be also necessary to be able to interpret as well as produce a select set of the basic network requests.

3 VHDL Implementation Issues

One of the biggest challenges in designing and implementing the PML has been to keep simulation run-time to a minimum, while providing a wide assortment of modeling capabilities. The LWP is an important piece to the PML, allowing the designer to sacrifice high-level O.S. functionality for greater simulation speed. (as much as two or three fold improvement). But with high-grained modeling, such a sacrifice may not be possible. In those cases, especially with multiprocessor systems, the efficiency of the PML is still an issue. Nobody wants to wait an additional few minutes for their simulation to finish (or even an additional hour for large, compute intensive simulations). This is where extra, careful effort has to be made, to overcome the limitations of VHDL.

3.1 Memory vs. Modularity

One issue to overcome is the trade-off between memory size of the simulation (which directly effects simulation run-time) and the usability, or modularity, of the library. VHDL configurations provide a means of abstracting levels of the model, making the model more modular through the creation of architecture hierarchies (a clear advantage in the language). Unfortunately, configurations also use up the most memory. Depending on the language implementation, the VHDL simulator will allocate separate chunks of memory for different instantiations made with VHDL configurations, while instantiations made within VHDL architectures will utilize shared memory whenever possible. In fact, the memory load for VHDL configurations is significant enough to dissuade VHDL programmers from using configurations at all unless absolutely necessary. Traditionally, VHDL models contain flat configurations, or a shallow hierarchy of configurations (one or two levels). The PML is by nature multi-layered, containing several levels of subcomponents for the detailed processor model. The issue thus is when to use configurations to represent a subcomponent in the PML, and when to embed the subcomponent instantiations inside VHDL architectures.

The resolution to this issue centers around the perspective of the PML user. Those subcomponents that require the PML user to manipulate parameters, e.g. the processor hardware kernel, are instantiated via a configuration. This gives the user a very straight-forward and readable format

with which to define the parameters. Those subcomponents that the user will most likely not touch (where much of the PML implementation is buried) are instantiated within the VHDL architecture from the layer above. The PML could have been implemented so that the processor model contained only one, monolithic configuration at the top-most level, which would save the PML some run-time memory. Usability, however, was deemed too important to warrant such an approach.

3.2 Instantiating Software Tasks

During the development of the PML, the instantiation of the software tasks also became an issue. In the PML, software tasks are represented by different, user-defined architectures of a special software task entity. Each software task is then assigned an index to an array of signals that can communicate with the rest of the processor model. The collection of all software tasks is encapsulated within another VHDL architecture one level up, and this architecture in turn is instantiated in the architecture representing the top level processor model component. Originally, the number of software tasks for a processor model was a fixed constant n , and a VHDL `generate` statement (present in the mid-level architecture) would loop over the range 1 to n to instantiate the individual software tasks. This approach was too costly, though, since most of the software tasks were never active. Many task components would be instantiated, and take up valuable space, even though they were never used. Inactive tasks were assigned a default architecture that essentially did nothing.

A part of the solution is to create a flag for each of the n software task, which would signal the `generate` statement if a software task needed to be instantiated or if the task could be ignored (this actually requires two levels of `generate` statements, one embedded within the other). The solution is not complete, however, because of restrictions put on the signals that connected the software tasks with the rest of the processor model. The indices for the signal array had to be sequential from 1 to m (where m is the number of active tasks). The PML could mandate that the user provide the correct index for the software task's signal, but such a requirement would be a nuisance to the user, and could potentially be a source for error.

The subsequent plan was to create a function for the mid-level architecture, a function that accepts a collection of boolean flags, determines which software tasks were active, and then returns a sorted array of signal indices. This function could not be used in the mid-level architecture, though, because VHDL `generate` statements can only take as arguments either literal values or generics, not constants, variables, or expressions. Thus, creating a function to be used within the `generate` statement of the

mid-level architecture was not an option.

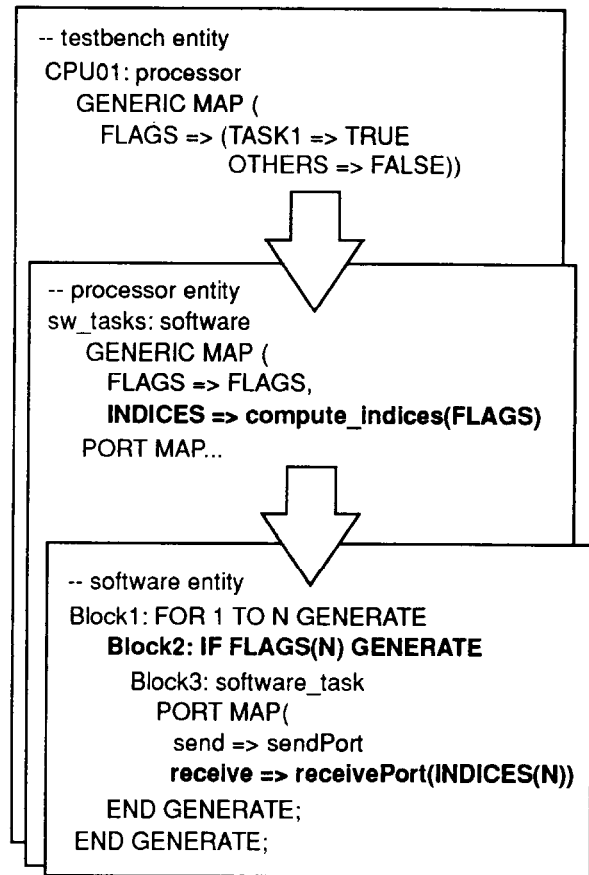


Figure 5: Software Task Instantiations

The mapping of such a function to a VHDL generic is allowed, however. The statement “GENERIC MAP (INDICES => compute_indices(FLAGS))” is perfectly legal. The final solution to the software task instantiation problem is to use a function within a generic map statement at the top-level processor architecture, which passes an array of sorted signal indices down to the mid-level architecture. At the mid-level architecture, the indices become the generic values, which then can be used within a `generate` statement to instantiate only those software tasks that are flagged to be active. See Figure 5. This maneuver, although rather involved, saves a significant amount of space and simulation run-time, without putting any additional burden on the user.

3.3 Signals and Strings

Signal assignments is another place where the PML is able to save simulation run-time. Communication between PML components is transmitted over a performance token bus that is consistent throughout the entire model. This performance token is essentially a record containing a

dozen fields, used to regulate token passing. Normally, one would assign a value to the token bus with a one line assignment statement:

```
Tout <= the_token
```

In most situations, though, such a statement is overkill, because only a few of the token's fields have changed. During a signal assignment statement, VHDL will create a transaction for every field in the signal definition type, whether that field's value has changed or not. Thus, for a signal record with twelve fields, at least twelve separate VHDL transactions will transpire (possibly more, depending on the bus-resolution-function). Each transaction requires simulation time, so a transaction representing a signal value that does not change is a waste. Whenever possible, the following type of signal assignment statements are used instead.

```
Tout.size <= the_token.size  
Tout.state <= the_token.state  
Tout.value <= the_token.value
```

Chopping up the token bus assignments does create more verbose code, however this is code the PML user would not likely have to look at.

A similar, unsolved problem occurs with character strings. The PML uses strings as an easy mechanism to define the source and destination components of performance tokens. When transmitting strings over a bus, however, VHDL considers the string to be a collection of individual characters, so the simulation creates a transaction for each transmitted character. This can potentially create a noticeable delay for simulation with a high volume of token passing. The only solution is to keep the size of character strings to a minimum, until a version of VHDL arrives that treats character strings in signals as atomic values.

The string mechanism in the PML that identifies the source and destination of performance tokens was not always an easy mechanism to use. String support is often a problem in VHDL. Commercial string packages for VHDL are often too large and burdensome to use effectively, while creating in-house string packages takes a lot of time and effort. Fortunately, the PML was able to utilize a rigorous string, manipulation package, which includes regular expressions, written by Lt. Greg Peterson of Wright Labs [5].

4 Software Design and VHDL

Previously, it was discussed how the PML supports hardware/software co-design. As explained earlier, this design approach can be accomplished through a variety of different scenarios. Regardless of which of these approaches is employed to capture system performance and behavior, VHDL is ultimately used to represent and analyze soft-

ware designs. Although an essential issue remains: does VHDL have the expressive capability and accuracy to efficiently capture software design?

VHDL, in general, has the capabilities and the basic requirements for software modeling. VHDL is a fully expressive language that can capture nearly all the intricacies of a software design. Honeywell has had very good results modeling complex software system using these techniques. However looking at the big picture, some issues still remain. First, for the software engineer who should certainly be part of this process, VHDL is yet another complex language to become proficient with. It also represents one and possibly two additional translation steps in the software development process (This is the same rationale for using VHDL based design all through the system/hardware development process, from performance to gates). Certainly there is a risk that some important aspects of design can be lost or modified in these translations. Even in developments with Ada the design will require non-trivial manual translation.

One solution that addresses the first issue is to remove VHDL from the software engineer's view of the design tool. On top of VHDL, build a library of software components and graphical tools for software model development. Providing performance level software components is the first step, and it will make the overall toolset much more widely accepted and utilized. Yet through our previous experience with software modeling, it is apparent that some functional aspect of the software design will need to be included to capture the fidelity of the overall system. Expanding the software library to include functional software model components can be likened to the development of a universal software reuse library. Bringing in functional capabilities to the library is likely to be even more challenging than the original PML hardware component library development. It is critical that these software model components be well designed and very well generalized such that they can be truly plug and play and act compatible with other library components.

Software libraries and graphical tools are very reasonable solutions, but more profound changes are needed to drive this approach into the software development process. To provide a simulation platform acceptable to software engineers, it is likely that some adaptation to the VHDL language will be needed. Initially this could consist of a VHDL subset which eliminates from view much of the concurrency, which from the software perspective should not be visible, and must be supported by hardware models that exists in other parts of the overall system model. But ultimately, additional features need to be added to the VHDL language to improve efficiency in capturing software designs. Some areas might be dynamic memory,

variant records, global variables, flexible file I/O routines, etc. The net effect of this undertaking would be a language tightly coupled with the standard VHDL language, with hardware specific complexity removed, with enhanced software constructs and usability, and supported concurrently on the same simulator. Obviously the changes described above would require a tremendous effort by a large diverse group of contributors, an activity that would likely take years to incorporate even the most modest of suggestions. Nevertheless a serious need exists and will need to be addressed to really solve the hardware/software codesign and basic system analysis problem.

5 Acknowledgments

The PML was originally developed under the Cockpit Display Generator (F33615-92-C-3802) program from Wright Labs and has been significantly enhanced under the Rapid Prototyping of Application Specific Signal Processors (RASSP) umbrella. The RASSP program is a four and one-half year, \$150 million Advanced Research Projects Agency (ARPA)/tri-Service initiative intended to dramatically improve the process by which complex digital systems, particularly embedded digital signal processors, are designed, manufactured, upgraded, and supported. Under RASSP, this work has been done under the following contracts F33615-94-C-1495 (VHDL Hybrid Models), DAAL01-93-C-3380 (Lockheed-Martin ATL RASSP), and F33615-94-C-1495 (Omniview PMW). The PML is being commercialized by Omniview as the Performance Modeling Workbench.

The authors wish to thank Lt. Greg Peterson for the development of the VHDL regular expression package, Charles Buenzli and Jay Runkel for their commercialization support, and all the early users who have persevered. We also wish to thank Dr. John Hines, his staff, and Capt. Ken Runyon from Wright Labs, and Carl Hein and Dave Nasoff from Lockheed-Martin (ATL) for their support.

6 References

- [1] Rose, F., T. Steeves, and T. Carpenter, "VHDL Performance Models," *Proceedings 1st Annual RASSP Conference*, pp 60-70, Arlington, VA, August, 1994.
- [2] Hein, C., and D. Nasoff, "VHDL-based Performance Modeling and Virtual Prototyping", *Proceedings 2nd Annual RASSP Conference*, Arlington, VA, July, 1995.
- [3] Meyassed, M., R. McGraw, J. Aylor, R. Klenke, R. Williams, F. Rose, and J. Shackleton, "A Framework for the Development of Hybrid Models," *Proceedings 2nd Annual RASSP Conference*, Arlington, VA, July 1995.
- [4] Hancock W., J. Groat, T. Steeves, H. Spaanenburg, and J. Shackleton, "System Analysis of Graphics Processor

Architecture Using Virtual Prototyping", *SPIE's International Symposium on Aerospace/Defense Sensing & Control and Dual-Use Photonics*, Orlando, FL, April, 1995.

[5] Peterson, Lt. G., "A String Manipulation Package for VHDL", *VIUF Fall 1995 Conference*, Boston MA, October, 1995.