A Unified Component Modeling Approach for Performance Estimation in Hardware/Software Codesign

Grode, Jesper Nicolai Riis; Madsen, Jan

Published in:
Euromicro Conference, 1998. Proceedings. 24th

Link to article, DOI:
10.1109/EURMIC.1998.711778

Publication date:
1998

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Grode, J. N. R., & Madsen, J. (1998). A Unified Component Modeling Approach for Performance Estimation in Hardware/Software Codesign. In *Euromicro Conference, 1998. Proceedings. 24th* (Vol. 1, pp. 65-69). [711778] IEEE Computer Society Press. https://doi.org/10.1109/EURMIC.1998.711778

A Unified Component Modeling Approach for Performance Estimation in Hardware/Software Codesign

Jesper Grode and Jan Madsen
Department of Information Technology
Technical University of Denmark
2800 Lyngby, Denmark
email: [jnrg,jan]@it.dtu.dk

Abstract

This paper presents an approach for abstract modeling of hardware/software architectures using Hierarchical Colored Petri Nets. The approach is able to capture complex behavioral characteristics often seen in software and hardware architectures, thus it is suitable for high level codesign issues such as performance estimation. In this paper, the development of a model of the ARM7 processor [5] is described to illustrate the full potential of the modeling approach. To further illustrate the approach, a cache model is also described. The approach and related tools are currently being implemented in the LYCOS system [12]. Details and the basic characteristics of the approach can be found in [8].

1. Introduction

The complexity of todays electronic digital designs has, within a range of application areas, dramatically increased the need for new approaches to system design. System designers have to handle both hardware and software issues at the same time, making all components of a system work together in a feasible manner. In hardware/software codesign, the design of the hardware and software that, in the end, will comprise the whole system, is considered concurrently. This often proves to be a feasible approach that makes the final system comply with various design constraints such as tight performance constraints (digital signal processing, telecom), high demands for low power (portable applications), etc.

Figure 1 shows the important stages in the early phase of a codesign trajectory: The *selection* of components for a *Target Architecture* (TA) and the evaluation of this selection with respect to the application that should be implemented on the TA. When a feasible TA has been found, the codesign trajectory proceeds with synthesis, i.e. code-generation, hardware generation (high-level synthesis), etc. The evaluation of the TA can be with respect to different aspects such as performance estimation of the final system, power consumption estimation, estimation of memory usage, etc. In the final implementation of the system, the application is split up and implemented on different parts of the TA in order to meet the design constraints. Thus, an important aspect of the TA evaluation is to investigate the feasibility of each individual component. The evaluation of a component's feasibility relies on *component models*.

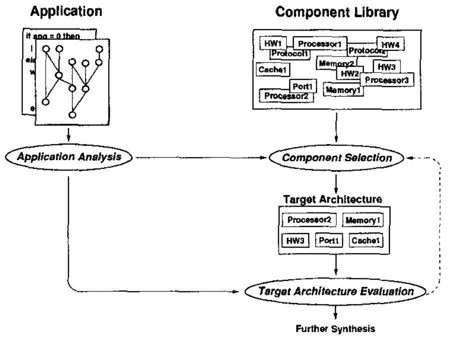


Figure 1. The early stages in codesign

This paper presents an abstract hardware/software architecture modeling approach. In this approach, different components like hardware accelerators (HW), micro-processors (SW), memories, peripheral units, etc., are described using Hierarchical Colored Petri Nets (HCPNs) [10]. This has several advantages over previous approaches:

High flexibility Using a unified modeling approach makes it possible to combine the models, thus yielding a high degree of flexibility in the modeling approach. This makes it possible to build advanced models from more simple models. E.g. incorporate a memory model in a micro-processor model, or incorporate a cache-model in a hardware accelerator model.

Higher accuracy Using HCPNs, the models can be made arbitrary precise (if inhibitor arcs are included in the CPNs, the modeling power of Petri Nets are raised to the level of Turing Machines [14]).

Furthermore, using a unified approach to describe the component models, the performance estimation scheme becomes simpler. This is illustrated in figure 2. If different approaches are used to describe different architectures

1089-6503/98 $10.00 © 1998 IEEE

(shown as grey boxes), different estimators will have to be used (shown as grey ellipses). However, when the unified modeling approach is employed, the estimators merges into one estimator.

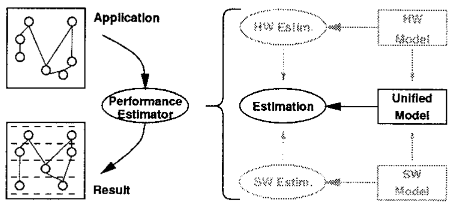


Figure 2. Performance estimation

Previous work within component modeling have traditionally been divided between HW and SW architectures, that is, between models of hardware accelerators and processor-like architectures. Traditional compiler techniques [2] and more advanced retargetable techniques [3, 4, 13] can be used to estimate component performance, but they are all targeted towards processor-like architectures. In [6, 7] a more high-level approach to performance estimation is taken. However, this technique is also guided towards processors. Techniques to describe hardware accelerator-like architectures obviously include languages like VHDL or Verilog. But they are too low level to describe the abstract behavior of processor-like architectures. In the TOSCA framework [15], the processor instruction set is modeled using VHDL, but unfortunately, the model is based on a generic instruction set which means that it can only model a given class of processors. Currently, the ACE environment [9] is used in the LYCOS system, but the approach has some deficiencies concerning the description of data-dependent behaviors such as pipeline stalls, cache misses, etc.

In the design environment ADEPT [11], component models are represented as CPNs and using VHDL. The CPN models are used for analytical approaches to analysis such as dependability analysis and the VHDL models are used for verification through simulation. However, the modeling approach is "hardware-near" in the sense that a predefined library of "hardware-near" basic building blocks is used to build more advanced models. Our approach differs in two areas: 1) it is truly unified because it describes software behavior in the same way as hardware behavior, and 2) when designing models, a top-down design approach is used instead of bottom-up as in the ADEPT environment.

To show the full potential of the unified modeling approach, this paper describes the development of a model of the ARM7 pipeline processor [1, 5] and of a cache model. It is, however, important to stress the fact that the approach can be used to model a wide variety of components often used in codesign target architectures. Section 2 will briefly describe the main aspects of the unified component modeling approach and how it is used for performance estimation. Details of the modeling approach can be found in [8]. The remainder of the paper will be devoted the description of the ARM7 processor model and the cache model.

2. The HW/SW modeling approach

A component model consists of two parts: an *implementation view* and a *service model*. The implementation view is the structural view of the component model and is not a topic for this paper. The service model gives information on which *services* the component provides. A *service* is a functional behavior that can execute a given application operation or function. The service model answers the following questions:

1. Which services are provided by the component? Examples of services are: 1) Simple instructions like addition and multiplication, 2) More complex behaviors like functions, 3) Other behaviors like AD/DA conversions, memory reads and writes, etc.
2. How is a service requested for execution and when will it finish execution? How many services can be requested at the same time?
3. How do concurrently running services interrelate? For instance, can one service delay another concurrently running service?

In order to answer these questions, it is necessary first to take a look at how applications are represented in the overall codesign framework. An application is represented as a hierarchical Control/Data Flow Graph (CDFG). The CDFG is implemented as a tree where the leaves are simple Data Flow Graphs (DFGs) and the internal nodes represent loops, conditionals and functional hierarchy. The nodes in the leave DFGs represent the basic operations of the application. As indicated in figure 2, the performance estimation is based on a schedule of the application. The performance estimator will use the following scheme (much like list-based scheduling) to generate a schedule of each of the DFGs in the application. These can then be used to produce a performance estimate for the whole application:

1. All operations in the DFG are successively requested to be executed by the services provided by the service model until all operations have been serviced.
2. In the current cycle, as many operations as possible are sought to be serviced without violating data-dependencies between the DFG operations and without exceeding the capacity of the service model. The capacity of the service model are the currently available services.
3. When all service requests in the current cycle have been made, the next cycle is entered by issuing a global update event. The services just requested have now started execution, and a new set of service requests can be made (repeat from 2).

Since the service model indicates when a service finishes execution, the performance estimator will be able to annotate each node in the DFG with the start-cycle and end-cycle (a service might take more than one cycle), thus producing a schedule which can be used as a performance estimate in the overall target architecture evaluation.

Some details in the above approach, especially the representation of time/cycles, will be highlighted as the ARM7 service model is presented in the following section.

3. The ARM7 Service Model

In order to better understand the ARM7 service model presented shortly, it would have been obvious to present the ARM7 architecture itself. However, due to limited space we will have to refer to the literature [1, 5] and instead, the architectural details of the ARM7 processor will be discussed as we go along.

3.1. Service Model Interface

The purpose of the Service Model Interface (SMINT) is to provide an interface for the performance estimator discussed above. The SMINT of the ARM7 is shown in figure 3. From the instruction set, the provided services are deduced and for each of these, a token is initially put on the place **Services**. This place holds tokens representing each of the services provided (in this case 16 which are listed in the dashed box). As it happens, most DFG operations can be serviced by instructions in the ARM7[1]. Only the `Division` operation is not directly covered by an ARM7 instruction (a solution to this problem is explained later).

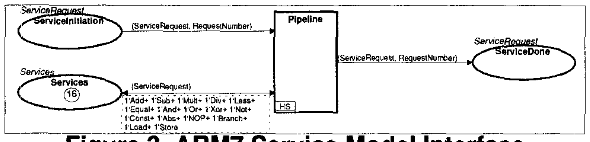

ServiceRequest
ServiceInitiation
(ServiceRequest, RequestNumber)
Pipeline
ServiceRequest
ServiceDone
(ServiceRequest, RequestNumber)
Services
Services
16
(ServiceRequest)
1'Add+ 1'Sub+ 1'Mult+ 1'Div+ 1'Less+
1'Equal+ 1'And+ 1'Or+ 1'Xor+ 1'Not+
1'Const+ 1'Abs+ 1'NOP+ 1'Branch+
1'Load+ 1'Store
HS

Figure 3. ARM7 Service Model Interface

The service model works in the following manner. The performance estimator will successively request operations to be serviced, simply by putting service-tokens on the place **ServiceInitiation**. If the same service is present as a token in the place **Services**, the transition **Pipeline** will become enabled. In a normal Petri Net, a transition is enabled when there are tokens on all input places. In CPN there is the additional enabling constraint that the tokens has to have the correct color. As an example, consider what happens if the service `Add` is requested in the service model in figure 3. A token `(Add,int)` is placed on the place **ServiceInitiation** to request the service[2] . Since there is and `Add` token on **Services**, the transition **Pipeline** is enabled and can fire. When the transition fires, the token `(Add,int)` is absorbed from **ServiceInitiation** and the token `Add` is absorbed from **Services**. In return, the token `(Add,int)` is produced by the output arc and placed on **ServiceDone**. Now, the performance estimator can see that this service has completed execution after one cycle. Note that the arc between **Services** and the transition is double-headed. This means that the token which is absorbed when the transition fires is immediately put back on **Services** thus modeling the fact that, in general, a new instruction can be started in each cycle on a pipelined processor.

[1]The processors instruction-set is far more complex than just to be represented by the 16 services shown in figure 3. However, the service model is used at an early stage in the design flow, and is used for *estimation*, not for actual code generation. Therefore, the key issue is to model the services required by any application DFG, thus finding the behavioral models of the services that can execute all types of DFG operations, if possible.

[2]The integer part of the token is used to distinguish multiple concurrently running services of the same kind).

It is important to note, that the SMINT always looks like in figure 3, no matter which component is modeled. This is what makes the approach unified. All component models look the same to the performance estimator. The only difference between different service model interfaces is in the number and types of provided services.

At this level of detail it seems that all instructions take one cycle to execute. This is however a poor model of the ARM7 instruction set and therefore the model is refined by letting the transition **Pipeline** represent a whole new CPN as described in the following section.

3.2. Service Model Implementation

The ARM7 processor is implemented using a five stage pipeline. The first two stages are the fetch and decode stages. Most data manipulating instructions like addition (Add) and comparison (Less) are executed using just one extra cycle, namely the execute stage. Store Register takes one additional cycle and Load Register and Branch takes five cycles in all (no delayed branch slot). These facts are used to construct the Service Model Implementation (SMIMP) shown in figure 4. Note, how conditional arc-expressions are used to determine the "path" of a given token. Also note, that the CPN shown in figure 4 is now the implementation of the transition shown in figure 3. Thus, we have obtained a more precise model of the services provided by the SMINT.

When designing a SMIMP, it is important to know how the service model is executed during performance estimation. In addition to the normal CPN execution rules [10], two rules apply for execution of a service model:

1. All enabled transitions fire on the next update event.
2. If two transitions are mutually exclusively enabled (only one of them can fire), the transition topologically closest to **ServiceDone** is chosen to fire.

In figure 4, the three transitions **Exe**, **Sto** and **Reg** all require an `e` token from the place **IBus**. This means that if for instance a `Load` service is on **ES**, enabling **Sto**, and a token on **DE** is enabling **Exe**, it will be **Sto** that fires on the next clock event because **Sto** is topologically closer to **ServiceDone**. This makes the `Load` service proceed in the pipeline, whereas the service on **DE** is stalled for one cycle. Thus, the enabling-place **IBus** models the internal bus which is used by all instructions in their execute phase. No two instructions can thus be in the execute phase at the same time (the execute phase of for instance a `Load` extends to after the **Reg** phase).

The place **ExBus** models the external in-data bus. This bus is used in each cycle to fetch the opcode of the next instruction, but it is also used to load data-values from memory. Thus, a fetch and a load also excludes each other. If the situation occurs, the load is allowed to continue, whereas the fetch is stalled for one cycle (according to rule 2 above).

Note how this approach makes performance estimation more accurate. Instead of statically determining an estimate for each service execution time, service execution time will

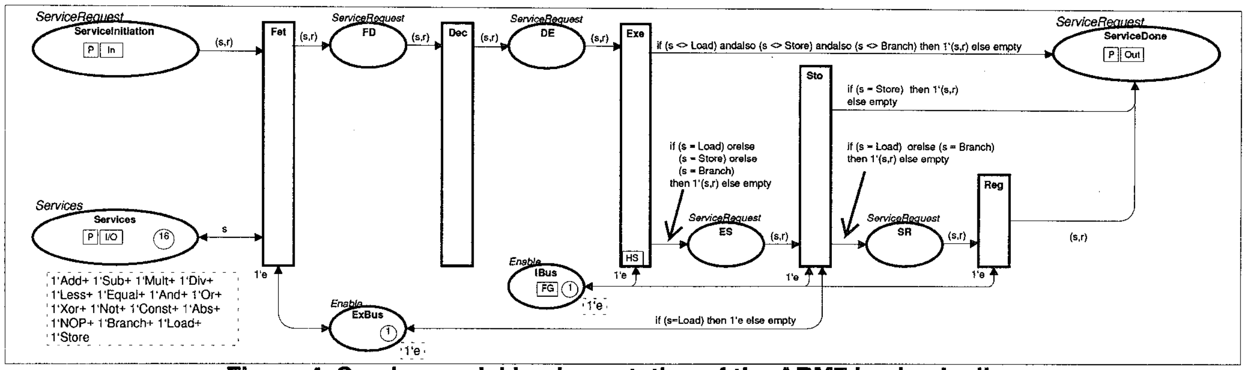

ServiceRequest
ServiceInitiation
P In
(s,r)
Fet
(s,r)
ServiceRequest
FD
(s,r)
Dec
(s,r)
ServiceRequest
DE
(s,r)
Exe
if (s <> Load) andalso (s <> Store) andalso (s <> Branch) then 1'(s,r) else empty
ServiceRequest
ServiceDone
P Out
Sto
if (s = Store) then 1'(s,r) else empty
if (s = Load) orelse (s = Store) orelse (s = Branch) then 1'(s,r) else empty
if (s = Load) orelse (s = Branch) then 1'(s,r) else empty
Reg
Services
Services
P I/O
16
s
1'Add+ 1'Sub+ 1'Mult+ 1'Div+ 1'Less+ 1'Equal+ 1'And+ 1'Or+ 1'Xor+ 1'Not+ 1'Const+ 1'Abs+ 1'NOP+ 1'Branch+ 1'Load+ 1'Store
1'e
ServiceRequest
ES
(s,r)
ServiceRequest
SR
(s,r)
(s,r)
HS
Enable
IBus
FG
1
1'e
Enable
ExBus
1
1'e
if (s=Load) then 1'e else empty

Figure 4. Service model implementation of the ARM7 basic pipeline

in this approach depend on the basic behavior of each service and on the *order* in which services are requested. In the ARM7 model, a Sub takes three cycles, but if the Sub request follows a Load request, the Sub service will take five cycles (three basic cycles plus two stall cycles).

Two services are not modeled correctly in the model in figure 4. This is the Multiplication and Division service. In the ARM7 processor, multiplication is implemented using a hardware implemented Booth multiplication algorithm which occupies the execute stage for at most 4 cycles (worst-case). Division is not implemented in the pipeline hardware and a call to the Div instruction will result in an undefined trap that can subsequently be handled by a software routine or attached co-processor (which can also be modeled by a CPN).

The modeling of multiplication on the ARM7 can be made more correct by refining the **Exe** transition. The implementation of this transition is shown in figure 5. All services except for the Mult service simply passes through after one cycle (the transition **MT1**). The multiplication service will however have to pass through all four transitions in the SMIMP in figure 5. Note, that when a multiplication is executing, no other instruction can enter the execute phase. This is ensured because the place **IBus** is a mirror of the place **IBus** in figure 4. Thus, the number of tokens on the two places is at all times the same.

Note, that when a transition is substituted by another CPN (**Pipeline** in figure 3 is substituted by the net in figure 4 and **Exe** in figure 4 is substituted with the net in figure 5), the substituted transition no longer "exists". This means that the substitution mechanism is a replacement strategy that implements hierarchy. A substituted transition has zero execution time.

3.3. Modeling a cache

The ARM7 service model can be extended to include service operand dependencies (instruction operands) so that for instance addressing modes can be modeled. However, due to limited space, a smaller example of operand dependencies will be outlined. To do this, a cache service model is used as an example.

A cache SMIMP is shown in figure 6. Note, that it is not complete. Only the cache-miss mechanism is shown, but the cache-hit mechanism is similar.

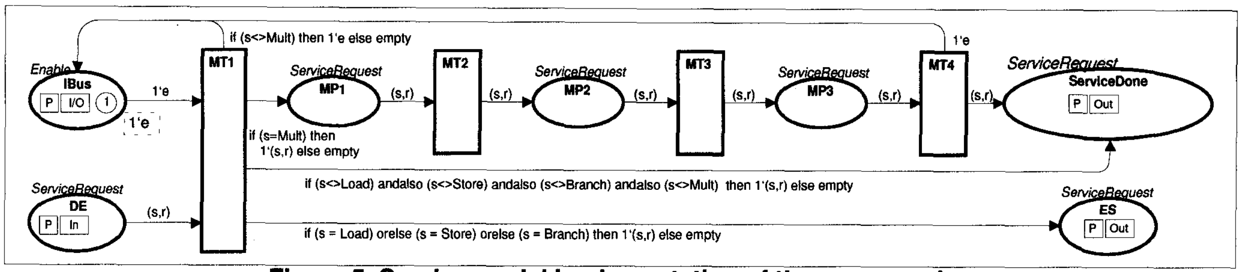
An operation in a DFG-like addition has two source operands. If these operands are allocated to memory, a Load service for each operand will have to be requested before requesting the Add service. Initially, all possible operands are placed on **NotInCache**. When requesting a Load service, the operand that should be loaded is used as an additional request token. To request a Load service in the cache model, the token (Load, int) is placed on **ServiceInitiation** while the token ("a", int) is placed on **OperandRequest**, assuming that the symbol that should be loaded is "a". The integer argument in both tokens should be the same so that the service request and the corresponding operand request can be related to each other.

If the operand is not on the place **InCache**, a cache-miss has occurred and the transition **M1** is enabled. When **M1** fires, several things occur. Firstly, the cache becomes temporarily disabled because the enable token from **CacheNotBusy** is removed. Secondly, the token "a" is removed from **NotInCache**. Thirdly, a token is put on **CacheMiss**, indicating a miss. Now, **M2** becomes enabled. The place **Filled** holds an integer-token that indicates how many free places are left in the cache (initially, 2 places are free). When **M2** fires, the number of free places is decremented by one, if it was not already zero. The requested operand is put on **InCache** indicating that the operand has been fetched from memory and is now in the cache. Also, if there were no more free places in the cache, a random operand (ro) from **InCache** is removed and instead put on **NotInCache**. The cache is re-enabled by putting an enable-token back on **CacheNotBusy**.

In the cache SMIMP, a miss only takes one extra cycle which is not realistic for most caches. To model miss-penalty and different cache-write strategies (such as write-back and write-through), the transition **M2** could be replaced by a CPN that models these aspects. Also, loads and stores (reads and writes) have different behaviors, which can also be modeled in the new CPN.

4. Conclusion and future work

This paper has presented a unified hardware/software architecture modeling approach using Hierarchical Colored



if (s<>Mult) then 1'e else empty
1'e
Enable
IBus
P I/O 1
1'e
1'e
MT1
ServiceRequest
MP1
(s,r)
MT2
(s,r)
ServiceRequest
MP2
(s,r)
MT3
(s,r)
ServiceRequest
MP3
(s,r)
MT4
(s,r)
ServiceRequest
ServiceDone
P Out
if (s=Mult) then 1'(s,r) else empty
ServiceRequest
DE
P In
(s,r)
if (s<>Load) andalso (s<>Store) andalso (s<>Branch) andalso (s<>Mult) then 1'(s,r) else empty
ServiceRequest
ES
P Out
if (s = Load) orelse (s = Store) orelse (s = Branch) then 1'(s,r) else empty

Figure 5. Service model implementation of the `Mult` service

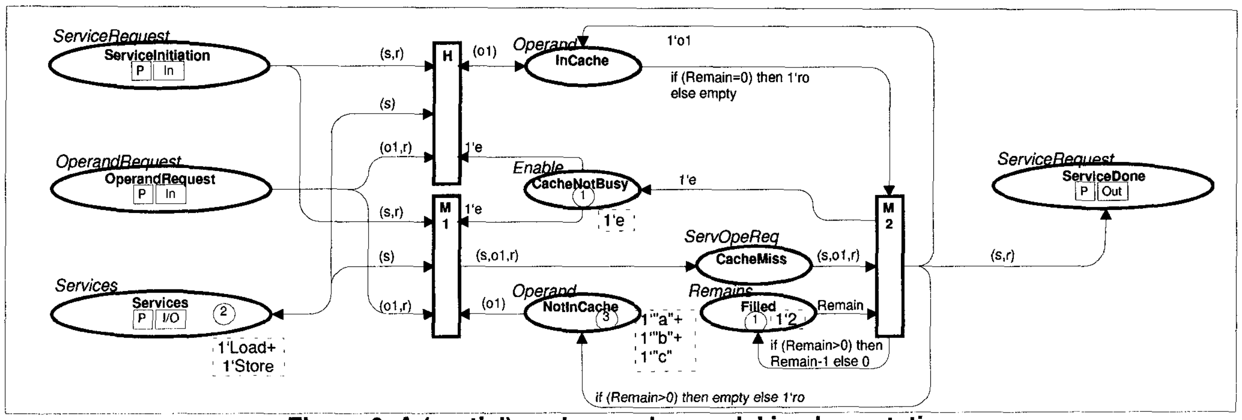


Figure 6. A (partial) cache service model implementation

Petri Nets (HCPNs). The modeling approach is used at the early stages in a hardware/software codesign trajectory where the designer has to make important decisions on target architecture composition and thus needs information on sub-component performance and behavior.

The paper has illustrated the unified modeling approach by showing the development of a model of the ARM7 pipeline processor and a cache. These two examples shows the strengths of the approach, namely more precise behavioral modeling capabilities than previous approaches and a high degree of flexibility using the hierarchical design techniques of HCPNs.

The approach and related tools are currently being implemented in the LYCOS system. The approach is currently targeted towards performance estimation. However, by attributing the transitions of the models with estimated power consumption or transition actions that emit assemblycode, a power estimator or code-generator can be obtained.

References

[1] Advanced RISC Machines Ltd (ARM). *ARM7 Data Sheet*, Dec 1994. Document number: ARM DDI 0020C.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language Version 4.1. Technical report, Lehrstul Informatik XII, University of Dortmund, September 1994.

[4] M. Freericks. The nML Machine Description Formalism. Technical report, Forschungsberichte des Fachbereichs Informatik, Technische Universität Berlin, 1991.

[5] Furber. *ARM System Architecture*. Addison-Wesley, 1996.

[6] J. Gong, D. D. Gajski, and S. Narayan. Software Estimation Using a Generic-Processor Model. In *European Design and Test Conference*, 1995.

[7] J. Gong, D. D. Gajski, and S. Narayan. Software Performance Estimation for Pipeline and Superscalar Processors. Technical report, Department of Information and Computer Science, University of California, Irvine, 1995.

[8] J. Grode, J. Madsen, and A.-A. Jerraya. Performance Estimation for Hardware/Software Codesign using Hierarchical Colored Petri Nets. In *HPC'98, Special Session on Petri Net Applications and HPC*, Boston, USA, 1998.

[9] B. Hald. *Architectural Synthesis Using Flexible Library Modules*. PhD thesis, Tech. University of Denmark, 1996.

[10] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer, 1992.

[11] S. Kumar, R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, and R. Waxman. ADEPT: A Unified System Level Modeling Design Environment. In *Proc. of the 1st Annual RASSP Conference*, Arlington, Virginia, August 1994.

[12] J. Madsen, J. Grode, P. Knudsen, M. Petersen, and A. Haxthausen. LYCOS: the Lyngby Cosynthesis System. *Design Automation for Embedded Systems*, 2(2):195 – 235, 1997.

[13] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publisher, 1995.

[14] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[15] D. Sciuto, S. Antoniazzi, A. Balboni, S. Milanese, and W. Fornaciari. The Role of VHDL within the TOSCA Hardware/Software Codesign Framework. In *Proceedings of the European Design Automation Conference*, 1994.