Model based approach to Hardware/Software Partitioning of SOC Designs

Pradeep Adhipathi

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Dr. James M. Baker, Chair
Dr. James R. Armstrong
Dr. F. Gail Gray

June 21, 2004
Blacksburg, Virginia

Keywords: System on Chip, Co-design, Partitioning, Hardware Modeling

Model based approach to Hardware/Software Partitioning of SOC Designs

Pradeep Adhipathi

(ABSTRACT)

As the IT industry marks a paradigm shift from the traditional system design model to System-On-Chip (SOC) design, the design of custom hardware, embedded processors and associated software have become very tightly coupled. Any change in the implementation of one of the components affects the design of other components and, in turn, the performance of the system. This has led to an integrated design approach known as hardware/software co-design and co-verification.

The conventional techniques for co-design favor partitioning the system into hardware and software components at an early stage of the design and then iteratively refining it until a good solution is found. This method is expensive and time consuming. A more modern approach is to model the whole system and rigorously test and refine it before the partitioning is done. The key to this method is the ability to model and simulate the entire system. The advent of new System Level Modeling Languages (SLML), like SystemC, has made this possible.

This research proposes a strategy to automate the process of partitioning a system model after it has been simulated and verified. The partitioning idea is based on systems modeled using Process Model Graphs (PmG). It is possible to extract a PmG directly from a SLML like SystemC. The PmG is then annotated with additional attributes like IO delay and rate of activation. A complexity heuristic is generated from this information, which is then used by a greedy algorithm to partition the graph into different architectures.

Further, a command line tool has been developed that can process textually represented PmGs and partition them based on this approach.

Acknowledgement

I would like to thank my advisor, Dr. Baker, for his constant support through out my studies at Virginia Tech. Apart from his valuable guidance in this research, he has helped me in sorting out a number of crisis that I had to face, both academic and administrative.

I would like to thank Dr. Armstrong for giving me the opportunity to work with him in this research and for his very useful comments and suggestions. He helped me grasp the fundamentals and encouraged me to improvise on it.

I must also thank Dr. Gray for agreeing to be a part of my committee and for patiently reviewing this thesis and giving his inputs.

This research was a part of a grant from Motorola. I would like to thank them for believing in our work and providing financial and material assistance for this project.

I would like to specially thank all my friends at Virginia Tech and at PSG Tech for the support and encouragement that they expressed in a variety of ways which helped me stay focused and successfully complete my thesis.

Last but not the least, I would like to thank my parents and my brother Sandeep for standing by me through these years. Their constant encouragement and moral support is the primary reason for whatever little I have achieved so far in my life.

Table of Contents

List of Figures

List of Tables

1 Introduction

In the past decade, we have seen the IT industry increasingly embrace a new paradigm called System On a Chip (SOC) design. This is a design methodology in which logically different system components, like ASICs, I/O devices, general-purpose processors and DSP processors, are realized on a single silicon chip. Advances in high-density fabrication, as well as cost effectiveness of the resulting systems, have contributed a great deal to this shift. As a result, however, the design of custom hardware, embedded processors and software that go into them have become very tightly coupled. Any change in the implementation of one of the components affects the design of other components and, in turn, the performance of the system. Hence, the traditional wisdom of designing and developing each component as a separate entity is no longer efficient. A more integrated approach is needed. This has led to the concept of hardware/software co-design and co-verification. The research in this thesis focuses on one specific aspect of co-design, namely partitioning.

The rest of this chapter introduces the co-design process in some detail. Two different approaches to co-design are widely recognized [1]. The conventional approach is a repetitive process favoring early partitioning of the system. The partitions are then modified and refined in each iteration, based on profiled information. The second approach is more modern and is model based, where the system is modeled and validated before technology assignment (partitioning) is done. The partitioning scheme and the tool developed as a part of this research apply to the latter.

1.1 Co-design Techniques

Though the co-design process applies to a wide spectrum of applications ranging from consumer electronics to plant control systems, the focus of this document is on its emphasis in embedded-SOC applications.

1.1.1 System on a Chip (SOC)

Wireless phones, PDAs, Hard Disk Drives and DVD drives are all examples of devices that use SOC designs. A System-On-Chip integrates Processor Cores (General Purpose Processors, DSP Processors, MPEG Processors), Embedded Memory, Analog Logic (PLL, A/D), Peripherals (UART, USB) and Custom Glue Logic (Standard Cells, Gates) into a single chip [2]. This leads to reduced overall cost, increased performance, lower power consumption and reduced physical size.
Figure 1.1 shows an example of SOC integration.

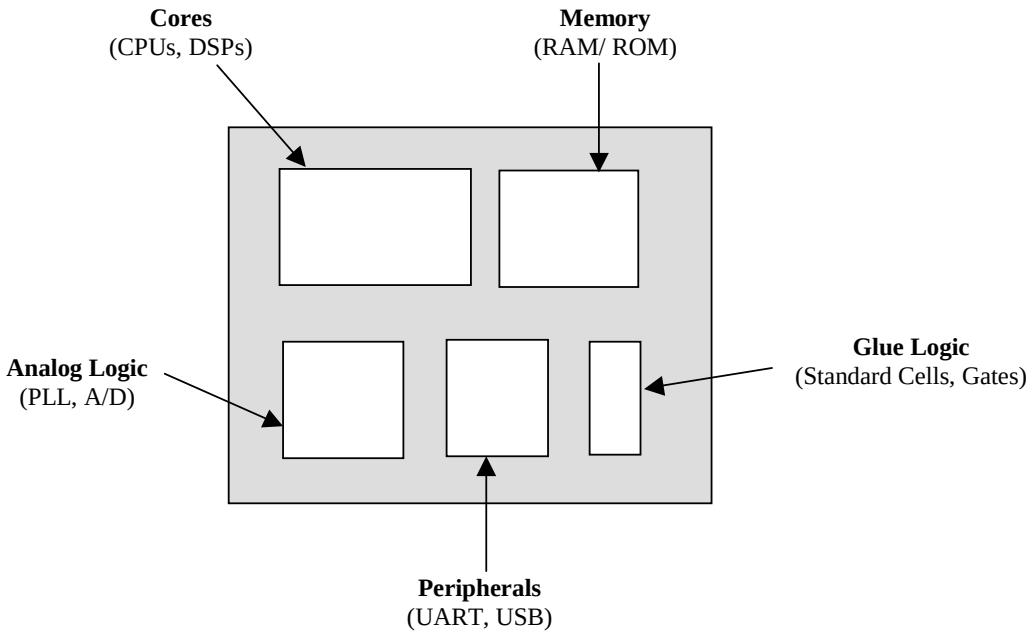


Figure 1.1 Example of a System On Chip Integration

1.2 The Conventional Approach

Traditionally, system design has involved manual intervention at various stages of the design flow. This not only affects the efficiency of the process, but also the order in which different steps are performed.

A typical design process would involve the following

- **Specification**
- **Partitioning**
- **Synthesis**
- **Integration**
- **Co-Simulation**
- **Verification**

Figure 1.2 illustrates this widely used design flow in the conventional approach to co-design. Each step is explained in detail in the sections that follow.

1.2.1 System Specification

The system design process starts with a specification of the system. The requirements of the system are typically specified in a non-formal language. However, there have been several attempts to formalize system specification to help in automation of the various steps that follow. ESTEREL [3], for example, is a Finite State Machine based specification language and is supported by co-design tools like POLIS [4]. System specification also incorporates Timing, Area and Power constraints.

1.2.2 System Partitioning

The system designer then uses the specification and his experience to make educated guesses on the performance of the system. Based on these guesses, he decides which part of the system will be implemented in hardware (as an ASIC) and which part in software. This step is called partitioning. It also involves writing the behavioral[1] description for the different parts of the system. The hardware part, for example, might be described using VHDL or Verilog and the software model using the *C language*. In addition, the interfacing logic, including any handshaking or bus logic, is also decided at this time.

[1] Behavioral here refers to functional logic and not "high-level description" as referred to in VHDL or Verilog text.

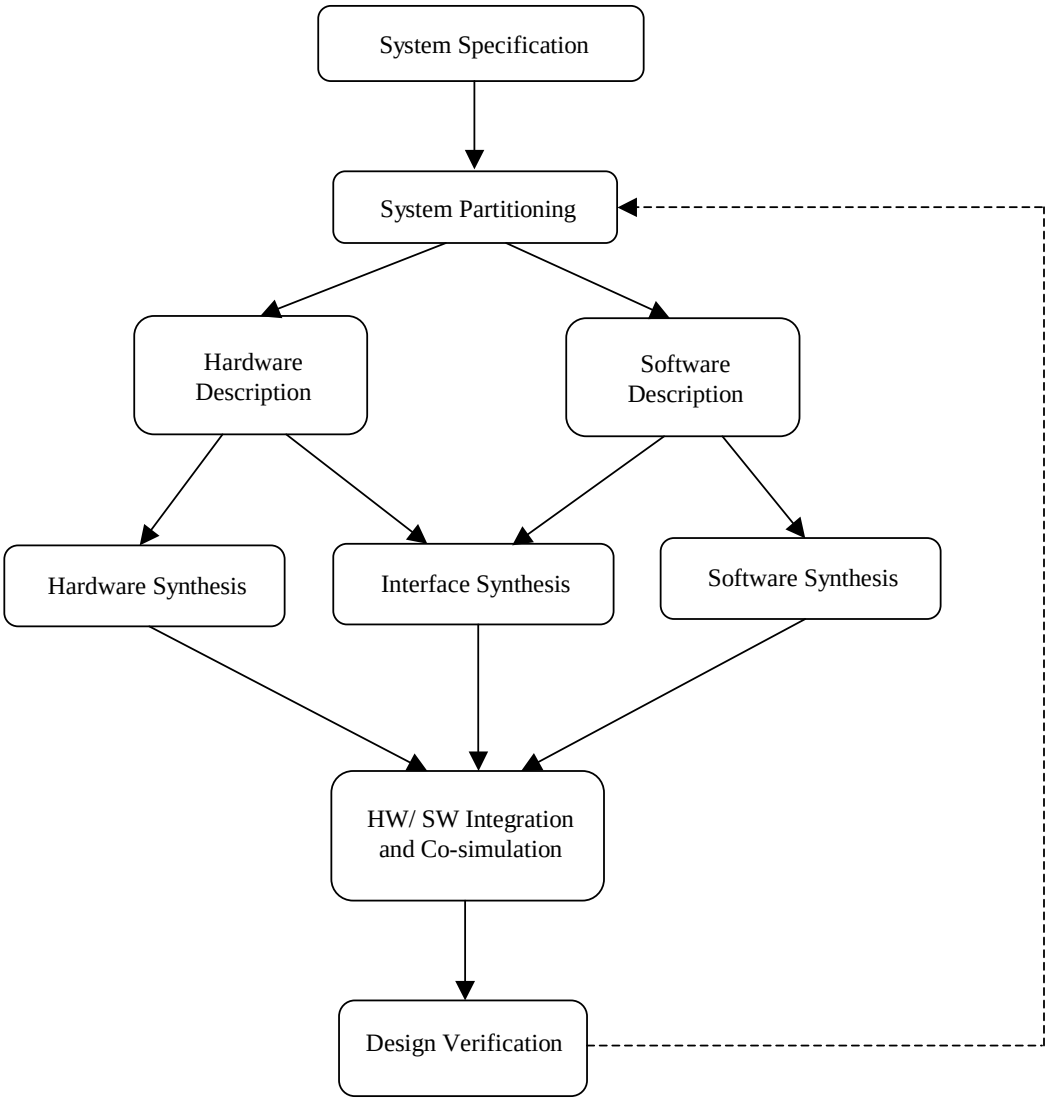


Figure 1.2 Conventional Approach to Hardware/ Software Co-design

1.2.3 Synthesis

Synthesis is the process of generating the physical model from the behavioral model. This translation can be performed with the help of synthesis tools like design compilers and cross compilers. Hardware synthesis tools, for example, can read hardware descriptions written in VHDL or Verilog format and generate mask layouts. Similarly, cross compilers compile programs written in a high-level language into the native instruction set of an embedded processor.

1.2.4 Hardware/ Software Integration and Co-simulation

Co-simulation is a difficult task and is a topic of recent research [5]. There are some commercially available co-simulation platforms that can simulate hardware and software synthesized models in an integrated environment[2]. But typically, designers either simulate the synthesized models separately and interpret the results or generate prototypes that can be simulated. The latter method, however, tends to be expensive, more so because this process may have to be repeated several times with modifications to the original design.

1.2.5 Design Verification

The results of co-simulation are verified against functional requirements and design constraints from the specification. The performance of the system is also validated at this step.

If the system does not meet the requirements, the entire process, starting with system partitioning, is repeated. Verification results may be used as hints to modify design decisions.

1.3 Model Based Approach

In the conventional approach discussed above, system partitioning is done very early in the design process. This reduces the flexibility of the designer and thus the efficiency of the final design. In this section, a more efficient scheme is proposed. Though the discussion is more general, it fits well to specific tools that were used as a part of this research.

The process is illustrated in Figure 1.3. As in every design flow, System Specification is the first step. The introduction given in the previous section holds for this section as well.

[2] Synopsys® CoCentric® System Studio and Agilent® Ptolemy Simulator

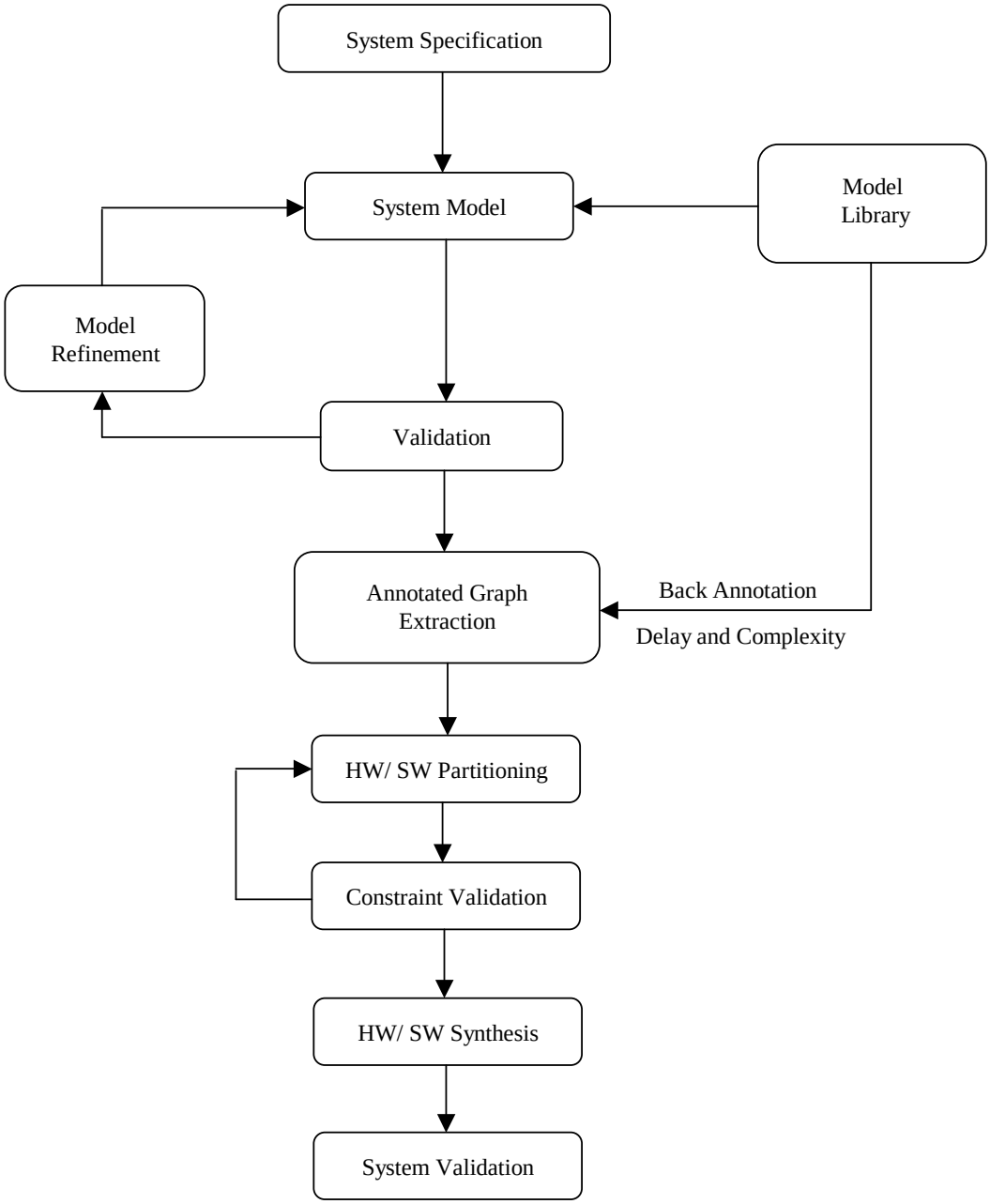


Figure 1.3 Model Based Approach to HW/ SW Co-design

1.3.1 System Model

The system model is a behavioral representation of the system written in a well-defined set of instructions. The amount of detail that a model abstracts is called the

granularity of the model. There are different levels of granularity. At the lowest level, a system can be modeled as an *algorithm* that just implements a truth table. Higher granularity is achieved by writing *register-level* and *gate-level* descriptions [6]. The Higher the granularity, the closer the model will be to the actual system. A good system model should have enough information to allow it to be simulated. The results of the simulation should be identical to the results expected from the final system.

The set of instructions, or the language in other words, that describe a system model has been evolving over several years. An ideal SLML (System Level Modeling Language) describes the exact behavior of any part of the system, independent of whether it is implemented as hardware or software. The recently proposed SLML, SystemC [7] [8], seems to satisfy this requirement and has been used in this research. SystemC is a set of libraries that extend the C++ language to model hardware. A SystemC program can be compiled with a standard C++ compiler and the generated object code can be used to simulate the model [22].

1.3.2 Model Library

The model can either be built from scratch or existing models can be reused. Models of various components of the system that have been thoroughly tested are added to a model library to facilitate reuse. As the library grows over time, design time is greatly reduced.

For SystemC, this library is a C++ library of classes and methods that represent a system component. The library can be linked to the code during compile time.

1.3.3 Validation and Model Refinement

Results of the simulation are used to validate the model. The validation here is largely functional verification and does not involve timing, power or area constraints. The output of the simulation is matched with the expected values.

The validation results may also be used to refine the model. For example, redundant or unused states and interfaces may be removed from the system.

1.3.4 Graph Extraction/ Annotation

This process is a unique feature of the model based co-design idea proposed by Dr. James Armstrong [9] and also forms the basis of the hardware/software partitioner developed in this thesis. A Process Model Graph [6], or PmG, is extracted from the refined system model and is a one to one mapping of the same. A PmG is a directed graph, where each node represents a process in the model and the arcs represent signals. The *process* models a concurrent task in the system. It is the basic construct in VHDL and is equivalent to *always* blocks in Verilog and *threads* in SystemC.

The PmG that is extracted is then annotated with important computation measures and timing details which are derived from the library and specifications. Figure 1.4 illustrates a delay and rate annotated process model graph.

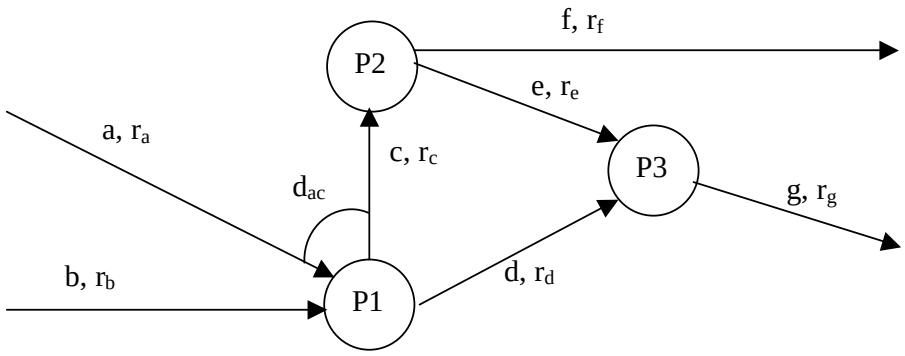


Figure 1.4 Rate and Delay Annotated Process Model Graph

In the above graph, P1, P2 and P3 are processes. Arcs are labeled (i, r_i), where *i* is the arc label and r_i is the activation rate of signal *i*. Activation rate is a measure of the rate at which a signal triggers the process. The delay between arc *i* and arc *j* is given by d_{ij}. More detailed analysis of this annotation is done in Chapter 2. Also note that graph extraction is a straightforward process and can be automated once the SLML is determined.

1.3.5 Hardware/ Software Partitioning

Hardware/ Software partitioning is also called Technology Assignment. The basic concept is the same as the system partitioning explained in the previous section. In this research, a methodology has been developed that uses the annotated PmG and partitions it based on a complexity heuristic. The idea is to fit as many nodes as possible into software as long the capacity of the embedded processor is not exceeded and the deadline constraints are met. As explained in chapter 4, this results in a more cost effective implementation.

This type of partitioning is similar to the problem of finding the largest connected sub-graph that meets certain conditions. This is an intractable problem in graph theory, so a greedy algorithm has been used to find a near-optimal sub-graph.

1.3.6 System Synthesis and Validation

The partitioned system is then synthesized into different entities, hardware or software. One reason why SystemC is evolving as a good choice for a SLML is its use of C/C++ language, which is the language used by many cross-compilers. Tools are being developed for behavioral synthesis of hardware from SystemC. Thus, the partitioned system, both hardware and software components, can be synthesized directly from the SystemC model with very little change. Co-simulation tools are then used to validate the synthesized system. This process can then be optimized further for area and power constraints.

1.4 Summary

A model based co-design methodology is proposed that involves rigorous modeling and verification before technology assignment. SystemC is used as the SLML. A Process Model Graph is extracted and annotated from the system model. The hardware/ software partitioner developed as a part of this research partitions the annotated PmG using a greedy algorithm and a complexity heuristic. Additionally, n-way partitioning (partitioning for multiple entities like general purpose processors, DSP processors, ASICs, etc.) and partitioning for hierarchical graphs are supported.

2 Previous Work

Hardware/Software partitioning has been actively researched as a part of the hardware/software codesign process for more than a decade. As chip densities have grown and component count multiplied, the problem has become increasingly complex. This thesis is an attempt to implement an algorithm based on a heuristic to solve the partitioning problem. There have been other recent publications that deal with the same issue. The rest of this chapter will summarize some of them.

The publication by Wayne Wolf [23] of the Princeton University explains how the concept of hardware/ software codesign evolved and how much it has matured over the past decade. Mr. Wolf also analyzes hardware/software partitioning and lists some of the open problems that are being currently studied.

Arato et al [24] give a formal analysis of the hardware/software partitioning problem. They prove that this is a NP-Hard problem and propose a genetic-algorithm based solution that produces near-optimal results even for very large systems.

Bhasyam et al [25] propose a dynamic programming framework for hardware/software partitioning which also incorporates the cost of communication delays between components of two different partitions. Their work attempts to find a minimum latency solution within finite resource constraints.

Patricia et al [26] describe their codesign tool CODEF that partitions components for SOCs based on their energy consumption. Their low-power allocation and scheduling algorithm uses energy and power models that produces partitions that have about 47% gain in energy.

There have been and continues to be several other proposals for efficient hardware/software partitioning. Each solution targets a certain objective like low latency or low power consumption. The implementation proposed in this thesis is based on a unique complexity heuristic and attempts to find a low cost solution that is also bound by time constraints.

3 Analysis of Process Model Graphs

The fundamental element in any system description is the *process*. It represents a set of activities. The activities within a process are considered to be sequential. But different processes operate simultaneously. That is, they execute at the same time, thus modeling the concurrent behavior of a system. Hardware Description Languages capture this behavior with different constructs. Verilog, for instance, uses the *always* and *init* blocks for concurrent modeling. Similarly, SystemC has *threads* and *methods* and the VHDL architecture has *processes*. A complete system can be thought of as one or more processes connected to each other by signals or wires.

Process Model Graphs (PmG) are directed graphs made of nodes and arcs, where nodes represent processes and arcs are signals. Additionally, the nodes of the PmG have ports to which the arcs are connected. The ports can be input only, output only or bi-directional. Thus, a process model graph is an abstract representation of a system that might be described in detail using a system level modeling language like SystemC. This close relation makes it easy to extract a PmG from the System Model. For more detailed discussion on PmGs, refer to [6]. Though the graph can be cyclic, the theoretical analysis in this chapter and the partitioning algorithm explained later are restricted to acyclic graphs.

It should also be noted that there could be different levels of detail when describing a system as a PmG. As an example, a system modeled at a higher level can be represented with components like multiplexers. A more detailed model would probably represent the multiplexer itself as a set of flip-flops. This level of detail is called the granularity of the model. Decreasing the granularity would mean more effort for the designer. The resulting PmG will be more complex with more number of nodes. But on the other hand, a detailed PmG allows for more flexibility in the partitioning process leading to more optimum partitions.

3.1 Formal Analysis of a PmG

Figure 1.4 showed an example of a PmG having three processing nodes with input and output signals connected to them. PmGs that are used in this research contain two special types of nodes, namely Source nodes and Sink nodes. All the input signals originate from the source nodes and output signals terminate at the sink nodes. Figure 3.1 shows the modified PmG with source and sink nodes.

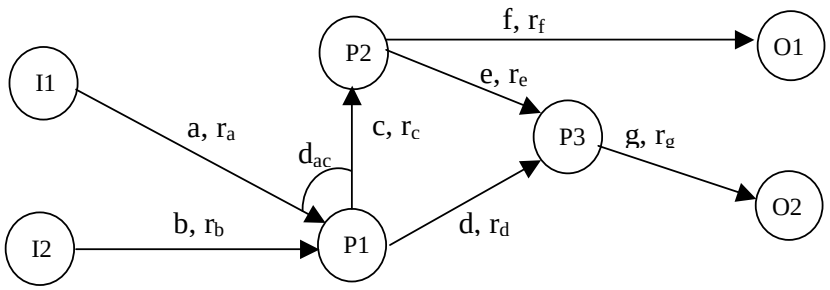


Figure 3.1 Process Model Graph with source and sink nodes

The following is a definition of the Process Model Graph from [9].

PmG is a digraph D with

D = {N, S}

Where N is the set of graph nodes (processes) and S is the set of arcs (signals).

Further,

$N = P \cup I \cup O$

Where P is the set of *n* Processing nodes, I is the set of *m* Source nodes and O is the set of *o* Sink nodes. Thus, each P_i (i = 1, 2, ..., n) is a vertex (node) representing a process, each I_j (j = 1, 2, ..., m) is a source, a node with in-degree = 0, and each O_k (k =l, 2, ..., o) is a sink, a node with out-degree = 0.

Also $s_i \in S$, (i = 1, 2, …, q) is an arc representing a signal.

3.2 Annotated PmG

A PmG in the abstract form described above, is of little use in partitioning. Certain properties of the processes and signals are valuable in making partitioning decisions. A PmG that stores these properties in its nodes and arcs is called an annotated PmG. The following are some of the attributes that are used by the Partitioner.

3.2.1 Activation Rate

This is the rate at which a signal triggers a process. It is an attribute of the signal and is annotated to both the input and output arcs in the PmG. There are several possible ways for calculating the activation rate, and they are explored in detail in the following chapter.

3.2.2 I/O delay

Delay is the property of the process and hence it is annotated to the node in the PmG. It is described as the delay between an input signal and an output signal. So, if a node has *n* inputs and *m* outputs, it will have *n x m* delay values.

3.2.3 Buffer Size

Buffer size is the amount of memory needed by the input and output buffers of a process. This value is annotated to the node.

3.2.4 Bus Width

This is the width of input and output signals and is annotated to the arc.

3.2.5 Power Requirement

The power consumed by each process is annotated to the node. This information is useful for partitions involving low power design.

The process of annotation is done at different stages of the design process. Data for some of the above attributes are readily available in the design libraries and can be determined at the time of graph extraction. Other information, like *interface delays*, vary dynamically based on the partitioning decision. These are *back annotated* after the preliminary partitioning is done. The back-annotated values are used for refining the partition further.

Figure 3.1 is also an annotated PmG. Here, arcs are labeled (i, r_i), where i is the arc label and r_i is the activation rate. Delay between arc i and arc j is written as d_{ij}. These labels are the annotated values.

3.3 Hierarchical PmG

Large systems are usually made of several subsystems and each subsystem is in turn made of several processes. It is sometimes very efficient to model such subsystems as a single node of the PmG. This node is then separately modeled as another PmG. A PmG that is made of such top-level nodes that represent other lower-level sub-graphs is called a hierarchical PmG.

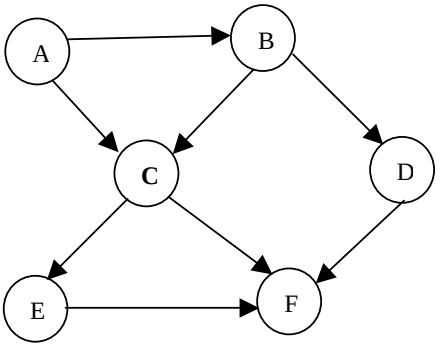
Figure 3.2 shows an example of a hierarchical graph. a) is the top level Hierarchical PmG. *C* is a hierarchical node. b) is a PmG that represents the node *C*. Note that the process names are qualified with the parent name. This module contains another hierarchical node C:D. c) is the leaf level PmG representing C:D. Such hierarchical PmGs can be useful for several reasons, some of which are discussed below.

3.3.1 Intractable SOC Designs

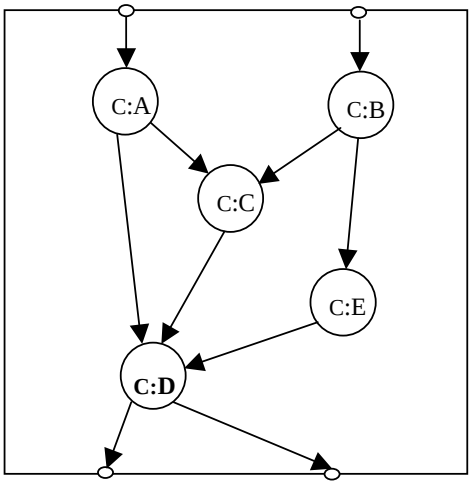
Generally, complete SOC designs are very huge and complex. Partitioning such graphs is sometimes intractable. Breaking down such systems into a hierarchical structure may provide a solution. The partitioning algorithm can be applied to the top-level graph in the first run and subsequent steps can operate on the hierarchical nodes that were selected in the previous run. To facilitate this, the node and signal attributes from the lower level graphs are extrapolated to their parent nodes.

3.3.2 Design Reuse

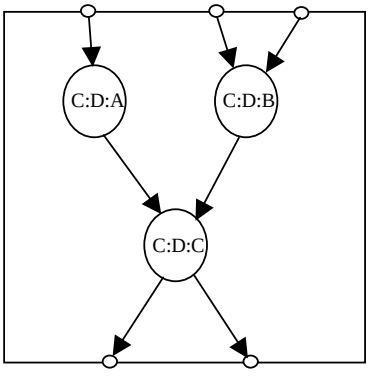
Several subsystems are readily available from legacy design libraries. Reuse of such models is very cost-effective. PmGs for often used subsystems can be kept in design libraries and used during graph extraction. Sometimes a component in the design is reused multiple times in the design. For example, a comparator can be used at various parts in the system. When such components are modeled as hierarchical nodes, they can be instantiated wherever needed. This makes the design more logical and easier to maintain.



a. *Top-Level PmG, with the Hierarchical Node* ***C***



b. *The Module representing Node C*



c. *Leaf level Module representing Node C:D*

Figure 3.2 Example of a Hierarchical PmG

3.3.3 Tightly Integrated Subsystems

Some subsystems can be very tightly integrated such that they cannot be split into software and hardware components. Hierarchical graphs provide a way to handle these situations.

3.3.4 Choice of Granularity

Hierarchical PmGs also give the choice of granularity in the partitioning decision. When flexibility in partitioning is needed, lower-level PmGs can be used. For quick partitioning, graphs at the desired abstraction provide a solution.

3.4 Textual Representation of the PmG

This section gives a very brief introduction to the file format used to represent a PmG. For the complete BNF grammar, see Appendix A. The format is specific to the partitioner developed in this research. Presently, the tool does not have graphical input/ output capabilities. Even when such features are added, a text format can be used to represent the graph.

3.4.1 PmG Identifier

The first word of the PmG is an identifier, either *PMG* or *HIER_PMG* and it indicates if the PmG is a top-level graph or a hierarchical module. For a hierarchical module, the next word is the Module Name followed by the module port mapping. This is shown in Figure 3.3.

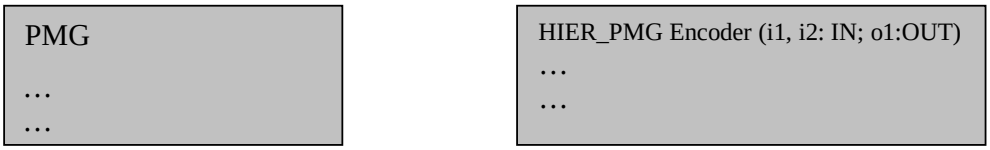


Figure 3.3. Code segment showing the PmG identifiers

3.4.2 Signals and Ports

The PmG has to be viewed more as a system than a graph. Therefore, the representation is closer to System Description Languages, like SystemC, VHDL and Verilog. The graph consists of signals and nodes. As mentioned earlier, nodes have input and output ports. The output ports drive the signals to which they are connected. More than one output port can drive a signal, though no special semantics have been defined for such a connection.

Signals with their activation rates (an attribute) are declared in the signal declaration area. Signal names are unique and can be any string of alpha numeric characters. Nodes are then declared one below the other, in any order. Hierarchical nodes are associated with their child module name. Port mapping can be either positional or associative (port names are contiguous numbers starting from one). Nodes also have delay information and other attributes mentioned earlier. An example of a top-level node is shown in Figure 3.4.

```
SIGNAL {
        i1      : 0.05;
        i2      : 0.23;
        o1      : 0.23;
        o2      : 0.05;
        o3      : 0.03;
};

comparator
{
        Port(i1, i2: IN; o1, o2:OUT);
        Delay {
        };
        Attributes {
        };
};

enc1 <Encoder>
{
        Port(o1: IN; o3: OUT);
        …
};
```

Figure 3.4. Syntax for Signal and Port definition

3.4.3 I/O Delay

Delay values are associated with each input port and output ports. The ports are represented as numeric values. If the delay between two ports is not specified, zero delay is assumed. But, if the delay values are specified for unknown ports, an error results. The Unit of delay may be MS, NS or PS representing microseconds, nanoseconds or picoseconds. Delays are specified for each entity (Software, ASIC) for which they are available. While partitioning for an entity, a node is chosen only if the delay value is available for that entity. Following is an example of a delay statement for a node with two inputs and two outputs.

```
Delay (HW) {
        1, 1: 10 NS;
        1, 2: 15 NS;
        2, 1: 12 NS;
        2, 2: 15 NS;
};

Delay (SW) {
        1, 1: 20 NS;
        1, 2: 25 NS;
        2, 1: 22 NS;
        2, 2: 25 NS;
};
```

Figure 3.5. IO Delay representation for a node in the PmG

3.4.4 Attributes

Attributes are used to store other annotated values, like Buffer Size, Bus Width, Power Dissipated, etc. They are written as *Name=Value* statements and may be used as a constraint while partitioning. The value should be numeric. Below is an example.

```
Attributes {
        BUS_WIDTH = 16;
        POWER = 0.2
};
```

Figure 3.6. Example of defining attributes for the PmG

3.5 Summary

A Process is the basic component that models concurrent behavior of a system. A Process Model Graph consists of *processes*, that have input and output ports and *signals* connected to them. A PmG is an abstract model of the system and can be easily extracted from a behavioral system model. When the attributes or properties of the processes and signals are stored in the PmG, it is said to be annotated. Some of these values can be extracted from the design libraries and others are back-annotated later in the design process. A PmG annotated with rate and delay information forms the basis of the partitioning algorithm proposed in this research. When PmGs are very large, the partitioning problem can be intractable. In such cases a Hierarchical PmG helps in reducing the size of the problem. They also provide a means for integrating tightly bound subsystems and facilitate design resuse.

4 Rate and Delay Estimation

Design Space Exploration is the process of dividing a system into hardware and software entities such that an efficient cost/ performance tradeoff is achieved. The partitioning idea proposed in this research is largely based on a complexity measure that is computed from the I/O delay of each node and its rate of activation. These values are annotated on the graph that is extracted from the system specification. The delay values are either obtained from design libraries or are estimated, while activation rates have to be computed for each system. Further, both hardware and software delay values are needed for each entity for which partitioning has to be performed.

Several methods have been proposed for estimating hardware and software delays. The accuracy of the estimates and the speed of the process depend on the individual methods. Sometimes these estimates can be educated guesses or at other times just upper bound values. Since performance estimation is not the main focus of this research, the following sections briefly describe recent research in this area and give references to related work. In the last section, methods for rate computation are explored and experimental results are presented.

4.1 Hardware Delay Estimation

Synthesizing and simulating the hardware model gives the best performance estimates for a node in the system. However, this is a very time consuming and inefficient process, considering that some of the nodes may not be implemented in hardware in the final system. If some of the models are being reused, accurate delay values can be taken from previously estimated values in the hardware library. When this is not the case, high-level estimation techniques are used which provide reasonably accurate and faster results.

High-level estimation typically falls under two categories, namely dynamic and static estimation [10]. Dynamic estimation involves simulation of an RTL model that uses design libraries specific to a hardware fabrication process. This method has the

advantage of producing more accurate results. But to create an RTL model, the entire system has to be converted from the system specification to a description that is closer to a hardware implementation. This process requires a great deal of time and effort. Static estimation on the other hand works at the system specification level and uses approximations or average values to produce reasonably accurate values. Most of the current work in performance estimation concentrates on static estimation. Following are methodologies proposed in some recent research and those that are already incorporated in co-design tools.

4.1.1 Boolean Function Method

In [11], the authors propose a novel method to compute delay associated with a combinational circuit from its high-level description. The functional description consists of ***n*** input, ***m*** output Boolean equations. The method uses the concept of a delay complexity measure, $M(f)$ of an ***n*** input, single output Boolean function ***f***, which is defined as a linear equation consisting of the size of a prime implicant and the number of prime implicants of ***f***. The intuitive reason behind this measure is that "the larger the number of literals in the prime implicant, the longer the time it would take for a function to be evaluated" and "the larger the number of prime implicants, the longer it would take to combine the results of the individual prime implicants" [11].

With this idea as the basis, an ***n*** input ***m*** output function ***f***, is first transformed into an ***n + log (m)*** input, single output function ***f'***, using a ***n x 1*** multiplexer. ***Log (m)*** is the number of control inputs to the multiplexer.

For several Randomly Generated Boolean Functions (RGBF), the delay measure $M(f)$ is computed as above. The actual delay $D(f)$ for the RGBFs is obtained from synthesis tools and a graph is plotted for $M(f)$ versus $D(f)$. The authors show that such a graph is a linear curve. Using this graph, the delay $D(f')$ for the corresponding $M(f')$ is found for any transformed function f'. The delay $D(f)$ for the original function f is then recovered from $D(f')$ after eliminating the influence of the multiplexer.

4.1.2 LOTOS Co-design Method

Another method of estimating delays is based on the LOTOS [12] system specification language. In this design flow the partitioning is done on a Process Communication Graph (PCG), which is similar to the PmG described earlier. Each node in a PCG is first decomposed into a Control and Data Flow Graph (CDFG) whose nodes represent basic data-dependent operations. The CDFG represents a flow made up of *control* nodes, which are synthesizable components from a design library. Each control node may further be associated with a Data Flow Graph (DFG). The DFG consists of data flowing to and from *operator* nodes. The response time for the operator nodes are fixed and are available from an operator library.

The I/O delay from every input signal to every output signal is then computed by summing the delay values of nodes in the maximum delay path. The methodology also takes care of Procedure calls, Conditionals and Loops using call times, probabilities and number of iterations respectively.

4.1.3 The QUEST system

QUEST [13] is an estimation tool that finds reasonably accurate area and delay values from high-level designs. The system design, in this case, is an RTL description of the logic. The basic idea is to implement a small subset of the design in the target technology and extract prediction parameters. This step is done only once and is applied to rest of the high-level design.

First, the RTL description is mapped into a technology independent logic network. A fully representative but small subset of the design is carefully chosen and implemented in the target technology. This implementation is optimized for area and time. It is assumed that the delay of basic gates is a function of fan out. A delay predictor function ***d(w)*** is obtained by solving a linear equation consisting of fan out ***w*** and generating function ***r*** corresponding to the implemented subset. The ***d(w)*** is restricted to a second-degree polynomial to speed up processing. This delay parameter is then used to solve the delay of any logic function to a reasonable degree of accuracy.

4.1.4 MUSIC based estimation

In [10], the authors use the MUSIC [14] codesign tool to implement an estimation methodology that uses both system level and RTL descriptions and combines both static and dynamic analysis techniques. A simple architecture is assumed for the hardware implementation where each VHDL RTL transition takes one clock cycle.

A high-level Specification and Description Language (SDL) is used for the System Specification. From the SDL, Basic Blocks (BB) are identified. A BB is a sequence of instructions that do not contain control and conditional operations. MUSIC then converts the SDL into a VHDL RTL specification for hardware implementation. The execution time for the RTL block that corresponds to each BB is computed in terms of number of clock cycles by statically finding the number of transitions.

4.2 Software Delay Estimation

The software part of a system is implemented as the object code for the target processor. Generally, when the abstract description of any part of the system is synthesized into software, code is first generated in a high-level language, like “C". The "C" code is then compiled using a cross compiler to produce the native object code. To estimate the delay for a piece of this code, timing information of the target processor is required. Timing is usually specified as the number of clock cycles per instruction (for each instruction).

As in the hardware case, accurate estimates of delay can be obtained if the object code is synthesized and simulated on a cycle accurate Instruction Set Simulator (ISS) for the target processor. Synthesizing and simulating the entire system is a tedious and time-consuming process. Methods that can produce reasonably accurate estimates from the high level descriptions rather than synthesized code are generally more useful and preferred. A great deal of research has gone into fast software performance estimation methods, a few of which are described here.

4.2.1 POLIS Based Estimation Methodology

In [15] the authors describe a method to get software performance estimates at different levels of design abstraction using the POLIS [4] hardware/ software codesign system. In POLIS, the system design is represented as a network of Codesign Finite State Machines (CFSM). The next level of abstraction for software is a set of *s-graphs* that are derived from the CFSMs. The *s-graph* is then converted to "C" code using a straightforward translation. Delay can be estimated either at the CFSM level or from the *s-graphs*. The latter is more accurate and is described briefly below.

The *s-graph* consists of nodes that represent source, sink, assignment and conditional constructs. Arcs represent the control flow. The control from the conditional node, for example, flows in different paths depending on whether the condition was true or false. The execution time **T** is defined as

$$\mathbf{T = T_{pp} + KT_{init} + T_{struct}}$$

Where $\mathbf{T_{pp}}$ is the time for entering and exiting the code block, $\mathbf{T_{init}}$ is the average initialization time for a local variable, **K** is the number of local variables and $\mathbf{T_{struct}}$ is the execution time for the structure of conditionals. $\mathbf{T_{pp}}$, **K** and $\mathbf{T_{init}}$ are constant for an architecture. $\mathbf{T_{struct}}$ is computed by traversing the *s-graph*. A Depth First traversal is used to find the maximum and minimum delay paths. The "C" code that is generated later has the same structure as the *s-graph*. Thus the execution time computed by this method is fairly accurate.

4.2.2 Source and Object Code Based Estimation

In [16], the authors present two processor independent techniques to estimate software delay. As the name suggests, the estimation is done starting with the source code (probably "C") and object code respectively. Both the methods use a processor independent set of instructions called the Virtual Instruction (VI) set. Each instruction in the VI set represents a class of instructions. That is, for every instruction in the VI set, there exists several tens of related instructions in the target processor. A processor basis file is created for each target processor, which maps every instruction in the VI

set to the number of clock cycles required to execute the related instructions in the actual processor.

In the source based estimation method, automatic annotation of the "C" code is first done. This is done by adding a function, _DELAY( ), for every statement in the source code. All the VIs that are needed to execute the "C" statement are passed as parameters. The function _DELAY( ) accumulates the corresponding cycle count for these VIs from basis file. Thus, execution of the annotated source code results in accumulation of cycle counts for the entire block of code. The accumulated value corresponds to the delay of the code. This method does not require a cross compiler or simulation on a native ISS.

While the source code based estimation method produces fast results, a disadvantage is that it does not take into account the optimizations performed by the compiler before the object code is generated. To overcome this, another similar method is proposed which works on the generated object code instead of the source code. Here the annotation is done for each instruction mnemonic in the optimized object code. Such an annotation can be achieved by tweaking the code generation part of the compiler. Execution of this annotated object code results in a fairly accurate timing estimate.

4.3 Rate Estimation

While delay is a common parameter in most of the recent work in design space exploration, activation rate is unique to this research. The following sections introduce the concept of activation rate and present two proposals for rate estimation of a process. The first method requires simulation of the entire system and produces fairly accurate results. An example system and the rate estimate obtained using this technique is shown. The second method is also based on simulation, but a smaller statistical model of the system is used.

4.3.1 Activation Rate Vs. Events

The input signals of a process can be broadly classified into two categories, namely triggering inputs and data inputs. Signals that activate the process execution like the clock signal, for example, are triggering inputs. Data inputs are available to the process when it is triggered. Data inputs can sometimes trigger a process, in which case, it is considered to be a triggering signal. The rate at which a process is triggered is the activation rate. The activation rate parameter that is used by the partitioning algorithm described in this thesis is associated with the signal rather than the process. That is, the activation rate is the rate at which each signal activates the process.

Most Hardware Description Languages have the concept of transactions and events. Every time a signal receives a value, even if it is unchanged from the previous value, a transaction is said to have occurred. On the other hand, an event takes place when the value on the signal changes to a different value. It can be clearly seen that,

$$\mathbf{r_t} \geq \mathbf{r_e}$$

where $\mathbf{r_t}$ and $\mathbf{r_e}$ are the rate of transaction and event respectively of a signal. A process can get triggered on any event or can be edge triggered, in which case a specific event (positive-negative or negative-positive) causes the change. Thus,

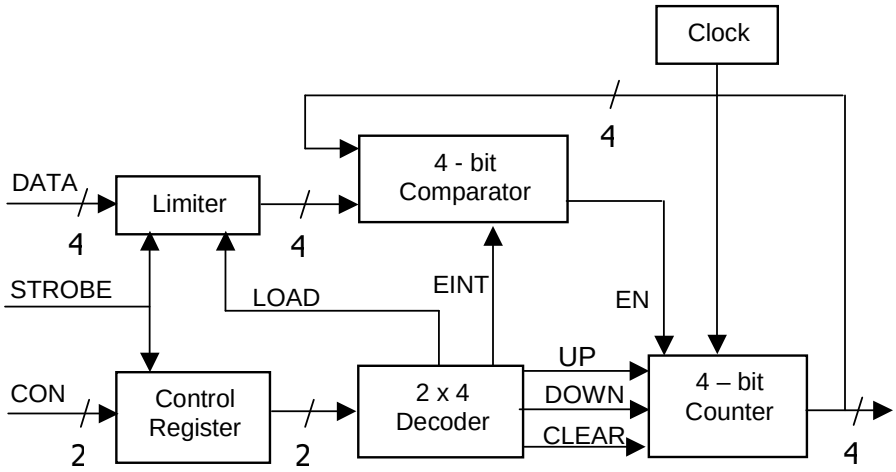
$$\mathbf{r_t} \geq \mathbf{r_e} \geq \mathbf{r_a}$$

where $\mathbf{r_a}$ is the activation rate of the signal.

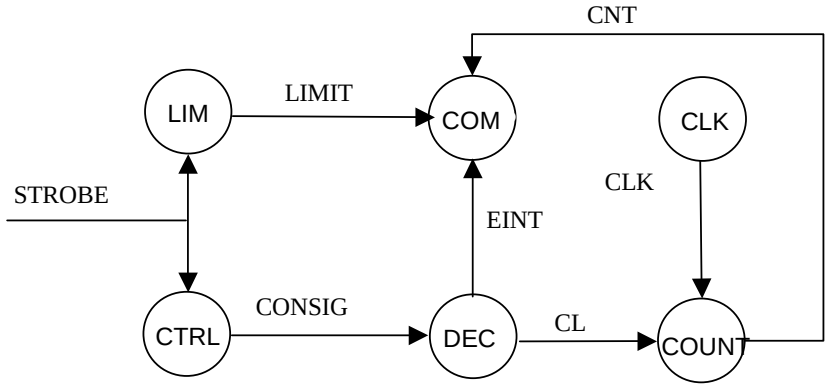
4.3.2 Event Trace Method

From the above argument, it can be concluded that event rate and transaction rate are worst-case measures for activation rate. Most of the high level simulation tools, including SystemC, provide some means to trace signal transactions and events. In the dynamic method, a representative input set is given to the system model and a high-level simulation is performed. A trace of the events on all triggering signals is generated. An automatic rate measurement tool then parses the trace file and measures the event rate on all signals of interest. This provides a good upper bound for the

activation rate. Figure 4.1 shows the block diagram and the PmG for a controlled counter system. An event trace was performed on the system. A "C" program was written to parse the file and generate activation rates. Table 4.1 lists the activation rate for each signal. A time varying plot of activation rate for the CLK signal is shown in Figure 4.2



a. Block Diagram



b. PmG with only triggering signals

Figure 4.1 The Controlled counter system

Signal	Activation Rate (events/ sec)
LIMIT	0.00
STROBE	3.42E4
CONSIG	1.39E4
CNT	6.16E4
CL	2.05E4
CLK	18.49E4

Table 4.1 Activation Rate Estimates for the Controlled counter

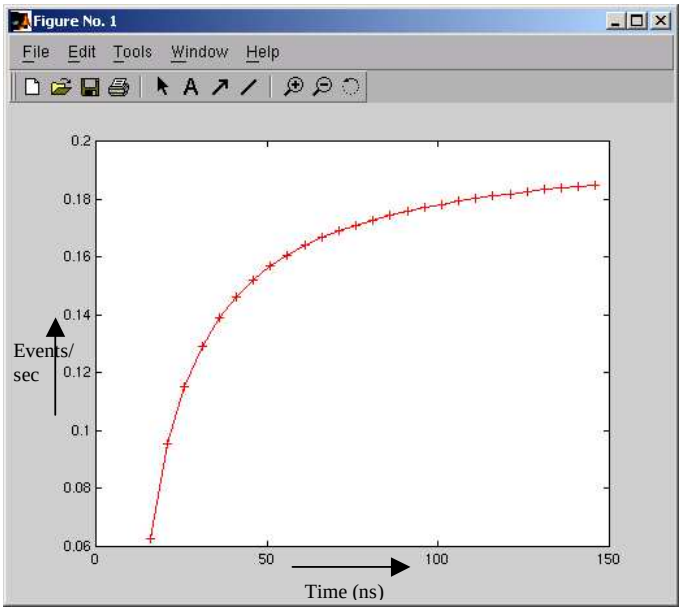


Figure 4.2 Activation Rate Plot for CLK signal

4.3.3 Rate Decimation Method

The above method produces accurate upper bound values. But simulating the entire system is an involved process. In the rate decimation model, simulation is performed on individual processes. The rate at which the outputs are generated with respect to the process activation is measured. This is the decimation rate for the signal. Now a system model is developed where the original processes are replaced by dummy processes that produce events on the output signals at their measured decimation rate. This system can now be simulated and the event rate can be measured using the previous method. In fact, a static estimation tool can be written which uses the decimation rate to predict the activation rates.

4.4 Summary

Performance parameters, particularly the execution time and the activation rate of each process, are an important part of the decision-making heuristic in the hardware/ software partitioning process. Estimating these parameters even before they are implemented on the target architecture is a challenging task. There have been several recent research efforts in the area of performance predictions. Using low-level methods, like simulation after synthesis, result in accurate measures, but they often require more effort and time. Therefore, most of the techniques propose using higher-level descriptions to produce fast and reasonably accurate values. Hardware estimation typically involves identifying basic blocks or Boolean equations and calculating the overall delay after reflecting the gate level implementation costs for these blocks. Similarly, the cost of executing an instruction on a target processor is used to measure the software delays.

5 Partitioning the PmG

There is often no single solution to the problem of hardware/software partitioning. Almost every process can be implemented in either hardware or software. As shown in [10], for a system specification with ***n*** processes, ***q*** different architectures (processors) and ***p*** different technologies (hardware/ software), the number of solutions is given by the exponential equation

$$N = \sum_{q=1}^{n} p^q S(n,q)$$

where, $S(n,q) = \sum_{i=0}^{q} \frac{(-1)^i \binom{q}{i} (q-i)^n}{q!}$ is the sterling number of the second kind.

Not all of these design solutions are feasible. Systems may have other constraints. For example, real-time systems may have hard deadlines that cannot be satisfied by some solutions. Further, each feasible solution will have a different cost/ performance benefit. The designer makes the final decision, which is typically influenced by several factors like performance, cost, availability of IP cores and existence of hardware and software components as libraries. While the partitioning process definitely requires a great deal of manual interaction, automating the elimination process and rationalizing the decision making can result in huge savings in terms of design to market time. With that in mind, the partitioning algorithm discussed in this research tries to find feasible solutions using a novel decision making strategy.

Software implementation is almost always cost effective compared to hardware realization. This is generally true because design-to-synthesis requires lesser effort in software than the equivalent hardware process. General-purpose architectures like DSP processors or RISC processors are available as IP cores at a cheaper price, which further reduces the cost. The partitioning methodology that is proposed here begins with the assumption that the entire system is implemented as a custom hardware. The algorithm then visits nodes (processes) in the *PmG*, finding those that can be implemented in software without breaking any design constraints. The following sections explain this process in detail.

5.1 Time Complexity

A node of the *PmG* may be triggered by one or more signals. The activity of the node begins when a signal triggers it and extends until the execution time of the node is complete. In the previous chapter, various methods for measuring the activation rate and the execution time were described. If the activation rate for a signal is high and the node it triggers has a long execution time then this makes the process *time-complex*. Thus, ***rd*** is a good measure for time complexity, where ***r*** is the activation rate for a signal and ***d*** is the delay from this input signal to the output of the process.

In embedded processors, the execution time is shared among several processes. If the time complexity of processes is high, only a few such processes can be mapped to the processor and still meet the overall system timing requirements. Since our intention is to implement the maximum number of nodes in software, a higher time-complex node is a candidate for implementation in hardware rather than in software.

5.2 Process Complexity

Process complexity is the computational complexity of a node. This reflects the complexity as a result of all signals incident on the node. The activation times of different signals that trigger a node are not always exclusive. But, the worst case is when the activation instants of incident signals do not overlap. Therefore, one measure of the process complexity is the sum of time complexities due to individual signals.

Consider the *PmG* shown in Figure 3.1. The process complexity of the process *P1* can be computed as below

$$Cp_1 = r_a d_{ac} + r_a d_{ad} + r_b d_{bc} + r_b d_{bd}$$

The same argument as for the exclusiveness of the *incident* signals also holds for the *output* signals. Firstly, all the output signals may not be triggered by every incident

signal. Secondly, the computation delay of different outputs may overlap. Thus, the equation given above is an upper bound of the process complexity.

5.3 Timing Constraints

Apart from process complexities, there are other design constraints that direct whether a node should be implemented as hardware or software. In real-time systems, signal paths can have hard deadlines. That is, for the system to operate as per design requirements, the delays of some paths in the *PmG* may be constrained to a certain value. In process control systems for example, the delay from the output of a temperature sensor to the input of a temperature controller may be critical for the operation of the unit and therefore may be constrained to a very small value. Thus, while trying to implement as many nodes in software, the critical paths should not be allowed to violate timing requirements.

5.4 Interface Considerations

If two processes that communicate with each other are implemented in two different entities, i.e. one in hardware and the other in software, then the inter-process signals between them will be routed through an interface mechanism. The most common interface is the Bus interface (for example, *PCI*). These interfaces are typically governed by protocols involving some form of handshaking. This further adds to the delay in the signal path.

Another problem with interfaces is that there is usually a limit to the number of signals available concurrently (the bus width). If more than the available signals operate simultaneously, then other means to negotiate the use of these signals have to be introduced. Such bus arbitration logic again leads to delays in the signal path.

Thus, it is clear that we would be better off if we try to cluster those processes that have a high degree of interaction between them into the same entity. The partitioning heuristic that is described below takes this factor into account.

5.5 Graph Partitioning Heuristic

The discussion in the above sections clearly point to a few parameters of a process that can help in choosing the best set of processes for implementation in a given entity. The heuristic that is derived from these parameters and explained below assumes that the nodes are being picked for implementation on a software entity.

The primary goal of our partitioning algorithm is to select the maximum number of nodes possible for implementation in software. Considering this goal along with the interfacing constraints, we conclude that the partitioned graph would be a connected sub-graph of the entire *PmG*. From the discussion earlier, a process is a candidate for software implementation if its complexity is small. Further, to find a large connected sub-graph, we intuitively reason that the degree of the node should be high. Combining these two parameters, we get the heuristic

$$h = \frac{Cp}{d}$$

where Cp is process complexity of node p
and d is the degree

Nodes are selected beginning with a small value for *h*. The selection is made iteratively until we reach the terminating condition.

5.6 Termination condition

Process complexity for software processes can also be thought of as the fraction of time that the processor spends in executing the process [9]. For all the processes to keep executing at the required rate, the sum of the process complexities should be less than one. That is,

$$C_{sum} = Cp_1 + Cp_2 + \ldots + Cp_n <= 1$$

Thus when C_{sum} reaches this threshold, the selection process is terminated. The nodes selected so far form the software partition.

5.7 Partitioning Algorithm

Figure 5.1 shows one implementation of the partitioning algorithm using the heuristic discussed above. Let S be the set of all nodes of a hierarchical PmG. Each node is visited and its process complexity and degree are computed and stored in the first phase. These values are assumed to be available and are referenced by Cp_i and D_i respectively.

Start:

Step 1: For **E = 1** to number of processors

Step 2: *Partition* (**E**)

Step 3: End For

Step 4: End

Partition (entity):

Step 1: For **i = 1** to **|S|**

Step 2: *SelectNode* (**i**, **Cp_i**, **p_i**);

Step 3: *ApplyConstraints* (**p_i**); if constraints met, goto *Step 4*;

Otherwise goto *Step 5*

Step 4: Add **p_i** to **P**; where **P** is the set of all partitions

Step 5: End For

Step 6: *SelectPartition* (**P**, **entity**);

Step 7: return

SelectNode (node_m, c_{sum}, partitionList):

Step 1: Mark node_m as Visited;

If (**c_{sum} > 1**) then goto Step 7; Otherwise go to *Step 2*

Step 2: Add node_m to partitionList;

Step 3: Let neighbor $\in \left\{ N_{node\_m}; N_{node\_m} \text{ is sorted in ascending order of } \frac{Cp_i}{D_i} \right\}$

Step 4: If **neighbor = Visited** or **Selected**, go to *Step 6*;

Otherwise go to *Step 5*

Step 5: SelectNode (**neighbor**, **c_{sum}+$Cp_{neighbour}$**, **partitionList**)

Step 6: if all **neighbors** visited, go to *Step 7*; Otherwise go to *step 3*

Step 7: return

ApplyConstraints (partition):

See the section "*Constraining the Partition*"

SelectPartition (allPartitions, architecture):

Step 1: Either Manually choose one partition or select a maximum size partition.

Step 2: For all nodes of the selected partition, mark the nodes

Selected for the given architecture

Step 3: return selected partition

Figure 5.1 Partitioning Algorithm, pseudo implementation

5.7.1 Constraining the Partition

This step ensures that the selected partition meets all the timing requirements of the system. The constraints are specified in terms of deadlines to be met by signals traveling through the system. Each deadline consists of a starting node, ending node and the maximum delay allowed between them. As a first step, the entire graph is traversed. For every signal connected between two nodes selected for different architectures, a dummy node is inserted between them. This is illustrated in figure 5.2. The dummy node is annotated with the interface delay.

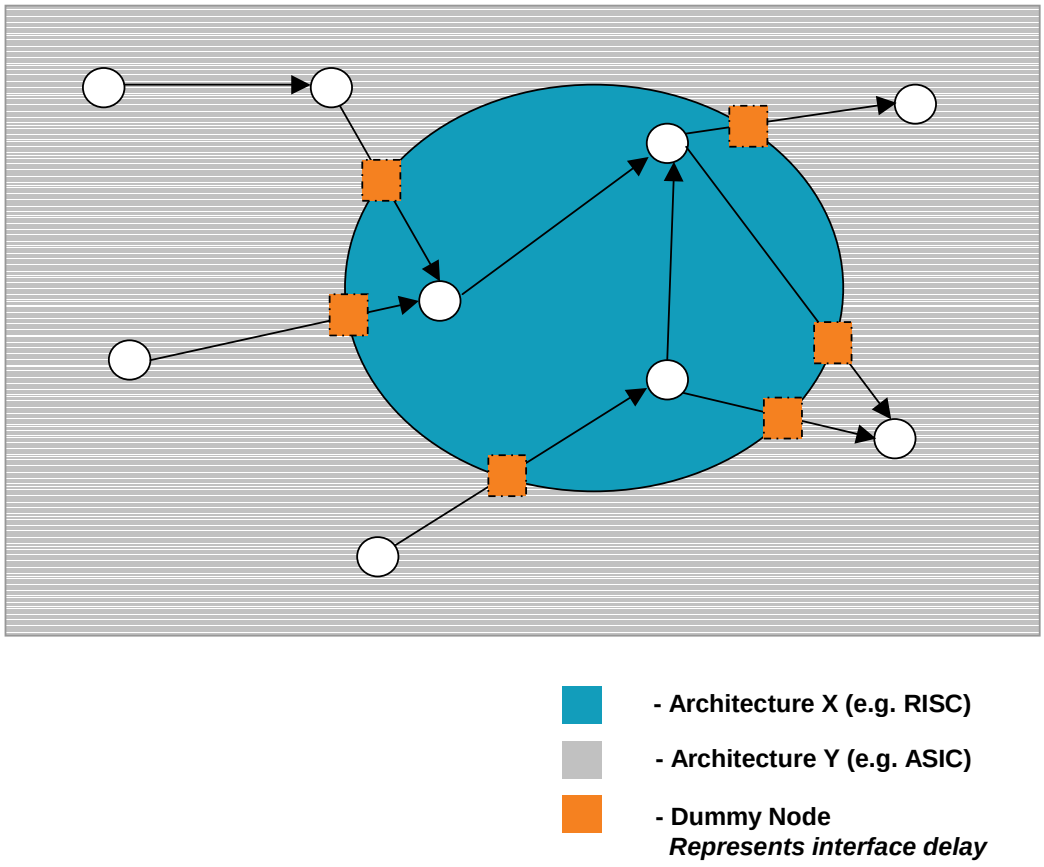


Figure 5.2 Representation of Interface Delay

For every deadline, all the paths originating at the start node and terminating at the end node are traced. The sum of the delays in the path is computed. The path with the longest delay is the *critical path*. If the delay of the *critical path* is less than the

deadline, this partition is selected. Otherwise the partition is not feasible and therefore rejected. The path tracing can be done either by a *BFS* (Breadth First Search) or a *DFS* (Depth First Search) algorithm. If the *PmG* contains cycles, path tracing becomes complicated. So, this partitioning algorithm does not support cyclic *PmGs*.

5.8 A Theoretical Example

This section explains the partitioning algorithm with a theoretical graph model. It is assumed that a graph (*PmG*) has been extracted from the system design and annotated with delay and rate parameters necessary for partitioning. Timing constraints and interface constraints have not been considered here since they are only applied on the partitions resulting from the algorithm and do not affect the partitioning process itself.

Figure 5.3 shows a PmG with five nodes. Each signal is annotated with its activation rate. The input-to-output delay for the processes is shown in Table 5.1. The complexities and the partitioning heuristic have been computed and presented in Table 5.2. Note that the *rate* and *delay* are in compatible units (i.e. cycles/millisecond and milliseconds). The algortihm attempts to find a partition starting at each node. From the resulting partitions, the one with the maximum number of nodes is chosen. The selection process is detailed below.

Starting at Node A:

The complexity of A is 0.4 which is less than the threshold value 1.0. So A is selected. Neighbors of A are B and C. Partitioning heuristic *h* for C is less than B. Therefore the algorithm attempts to map C to software. The complexity sum C_p is now 0.9, which is less than the threshold. So, C is selected. C's neighbors are B, D and E. First D is tried since it has the least value for *h*. But selecting D would cause the complexity to exceed the threshold. Then B and E are tried in that order. Both fail. Thus, the resulting partition includes nodes A and C.

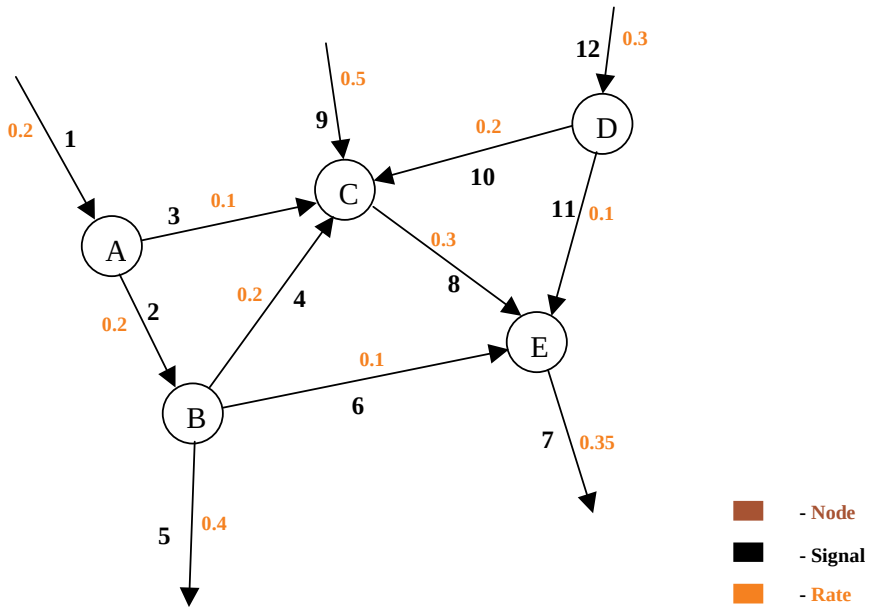


Figure 5.3 Rate annotated PmG

Delay in milliseconds (input signal, output signal)

Nodes				
A	1 ms ***(1,2)***	1 ms ***(1,3)***	-	-
B	2 ms ***(2,4)***	1 ms ***(2, 5)***	1.5 ms ***(2,6)***	-
C	0.5 ms ***(3,8)***	0.5 ms ***(4,8)***	0.5 ms ***(9, 8)***	0.5 ms ***(10, 8)***
D	1 ms ***(12, 10)***	1 ms ***(12, 11)***	-	-
E	2 ms ***(6, 7)***	2 ms ***(8, 7)***	2 ms ***(11, 7)***	-

Table 5.1 Input to Output delays for PmG in Figure 5.3

Starting at Node B:

B is selected with 0.9 complexity, which is less than the threshold value of 1.0. B's neighbors, C, A and E, are tried in that order (from low to high h). All of them fail the threshold test. So the partition contains only the node B.

Starting at Node C:

C is first selected since its complexity is less than the threshold. Now Cp = 0.5. Next C's neighbor A is tried first. Selecting A results in $C_p = 0.9$ (< 1.0). So A is selected. A's only remaining neighbor B is tried and fails the threshold test. Now C's remaining neighbors, D and E, are tried in that order. Both result in C_p greater than the threshold. So, the resulting partition has nodes C and A

Node	Degree (d)	Complexity (C)	Heuristic (h = C/d)
A	**3**	**0.4**	**0.13**
B	**4**	**0.9**	**0.22**
C	**5**	**0.5**	**0.10**
D	**3**	**0.6**	**0.20**
E	**4**	**1.0**	**0.25**

Table 5.2 Complexity and Partitioning Heuristic for Figure 5.3

Starting at Node D:

Node D is selected, resulting in a $C_p = 0.6$. D's Neighbors C and E are tried in that order. Both fail the test for threshold. So this partition only has node D

Starting at Node E:

Node E has a complexity of 1.0. Therefore the only possible node in this partition is E

From the five partitions found, the partition with A and C is the maximum sized partition found and is selected for implementation in the target architecture.

At first sight, it may seem that nodes not connected from each other but having a sum complexity of less than 1.0 are not selected. But actually, the partitioning algorithm is intended to be run more than once, each time selecting a cluster of nodes for implementation. This is continued till the threshold value of 1.0 is reached when the processing capacity of the target architecture is exhausted.

5.9 Summary

For a given design problem, the number of possible solutions can be exponential. Therefore, manually exploring the entire design space is sometimes impossible. To automate this process, an intelligent heuristic is needed. When the intention is to find the cheapest possible solution that can satisfy all the design constraints, a heuristic based on process complexities can find good hardware-software partitions. The idea is that a process that has longer execution times and requires repeated execution at a fast rate is more complex than one with a low rate of activation and shorter execution time. Using this as a heuristic, a greedy algorithm can be designed to find the maximum number of these less complex nodes that can fit into a given processor. If other constraints like interface delays and deadlines are taken into account, the greedy algorithm can form a good partitioning algorithm.

6 Partitioning Results

This chapter presents the results of applying the partitioning algorithm to a GSM (Global System for Mobile Communications) model. GSM is a typical application that involves hardware and software co-design. Since the system ends up as a portable/ handheld device, it is a good candidate for SoC design. GSM involves signal-processing components that can make use of DSP processor cores. However, inordinate delays in signal processing would result in loss of voice information, so some components may require custom ASIC solutions. This makes GSM a perfect example to attempt to find a good implementation using the partitioning algorithm.

For the system described in this section, the target architectures are the Motorola *StarCore 100* DSP development core and a custom *ASIC*.

6.1 Block Diagram

Figure 6.1 shows the block diagram for the GSM system. It is essentially made up of two parts, the transmitter and the receiver. The transmitter receives the voice signal, digitizes it, compresses and encodes it to protect from transmission losses, modulates the signal with a carrier and transmits it. The receiver does the reverse of the above. For more information on the different components, refer to [17].

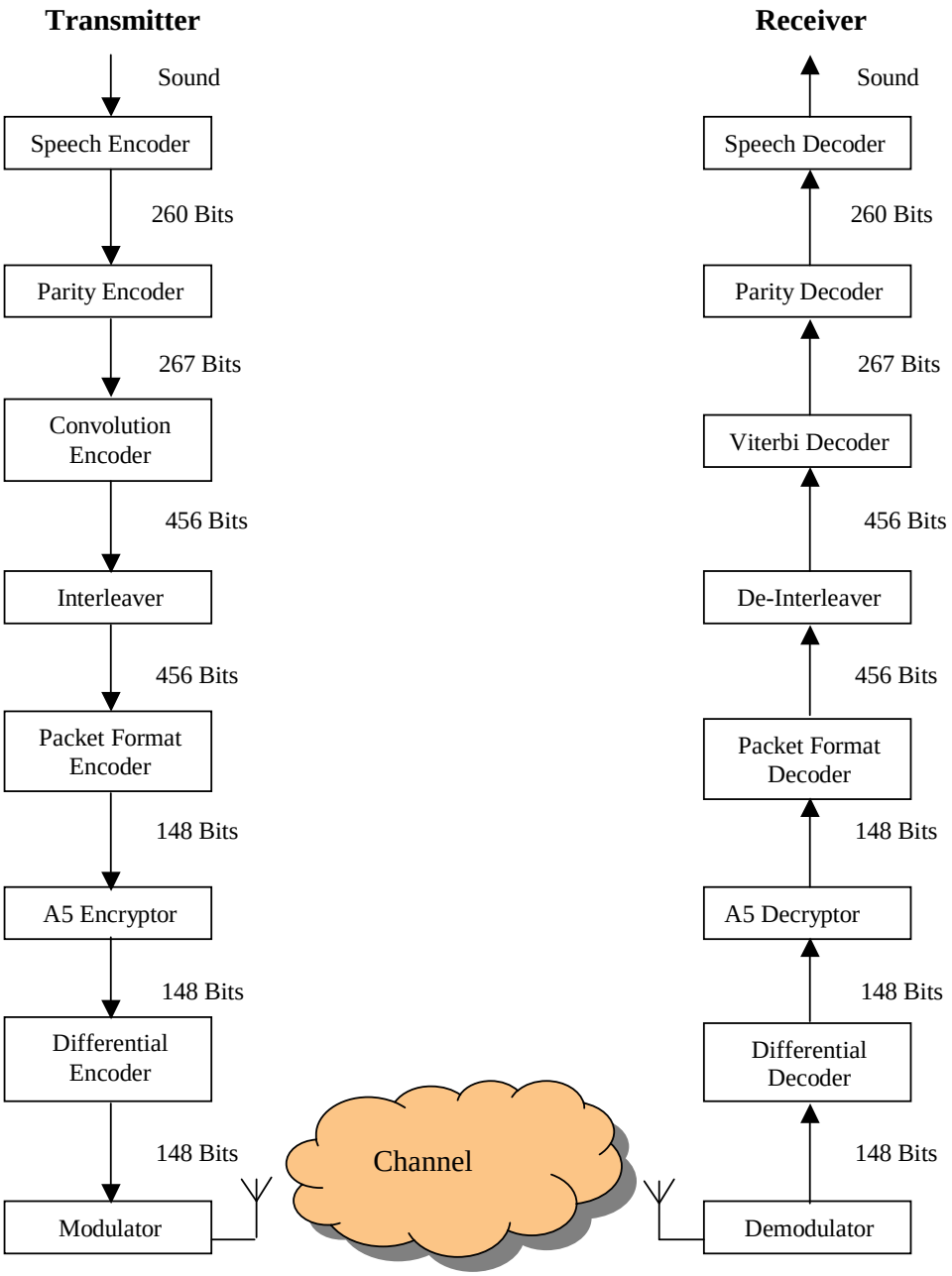


Figure 6.1 Block diagram of a GSM system

6.2 The PmG

Each module of the system was implemented as one or more processes in SystemC. Details of the implementation can be found in [17]. The processes were carefully chosen such that they can be implemented with very few changes in the StarCore processor. From the SystemC code, a Process Model Graph was manually extracted.

6.3 Hardware/ Software Delays

The first step in the partitioning process is annotating the PmG with delay and rate parameters. To measure the hardware delay, the SystemC code was synthesized using the Synopsys SystemC compiler with standard components. The delay between the input and output signals of each process was measured in terms of the clock cycles used. The frequency of the clock was chosen to be 300MHz, which is same as that for the StarCore processor.

The processes are then modified and implemented on a StarCore 100 instruction set simulator. The conversion process can also be automated. For a detailed analysis on transforming SystemC processes to StarCore's native C instructions, see [18]. The converted modules were then executed on the simulator with a block of test data. The execution time was recorded as the number of machine cycles.

Table 6.1 shows the execution times for each block of the GSM system in software and hardware implementations. Note that some values for the hardware implementation could not be found due to constraints in the System C synthesis tool.

6.4 Activation Rates

Since this is a synchronous system, a common clock triggers all the processes. Thus, the rate of activation has a fixed value of 300MHz.

Index	Module	Hardware Delay (ms)	Software Delay (ms)
1	Speech Encoder	-	4.1717
2	Parity Encoder	0.0087	0.0417
3	Convolution Encoder	0.0105	0.2384
4	Interleaving Encoder	0.0700	0.3532
5	Packet Encoder	0.0043	0.0226
6	A5 Encoder	0.0118	0.1461
7	Differential Encoder	0.0044	0.0306
8	Speech Decoder	-	1.6279
9	Parity Decoder	0.0155	0.0414
10	Convolution Decoder	-	44.6237
11	Interleaving Decoder	0.0756	0.3240
12	Packet Decoder	0.0053	0.0154
13	A5 Decoder	0.0118	0.1423
14	Differential Decoder	0.0039	0.0282

Table 6.1 Hardware/Software Execution times for different GSM blocks

6.5 Timing Constraints

Each module of the GSM system operates on a 20ms block of voice data, sampled at 8KHz. At the output, the delay between the first block and the second cannot be more than 5.1ms. If the delay exceeds this limit, it becomes perceivable by the human ear. Such a system becomes unfit for consumer use. The system has to be partitioned such that we meet this deadline.

6.6 The Partitions

The PmG is annotated with the delay and activation rates. Table 6.2 gives the process complexities for the system. Since the degree of every node is equal, that parameter can be ignored. With all the necessary data obtained, the PmG is then represented in a file format understood by the tool and partitioning is performed. This process follows the algorithm explained in the earlier example. That is, it starts with each node and

executes the greedy algorithm, until it reaches the threshold. So, starting with each node we get a partition. Each partition is now constrained using the timing deadline. From the partitions that succeed the constraints, the largest is chosen for implementation on the processor.

Process	Complexity
Speech Encoder	0.208585
Parity Encoder	0.002085
Convolution Encoder	0.01192
Interleaving Encoder	0.01766
Packet Encoder	0.00113
A5 Encoder	0.007305
Differential Encoder	0.00153
Differential Decoder	0.00141
A5 Decoder	0.007115
Packet Decoder	0.00077
Interleaving Decoder	0.0162
Viterbi Decoder	2.23118
Parity Decoder	0.00207
Speech Decoder	0.081395

Table 6.2 Process Complexities

From the complexities table, it is apparent that every process except the Viterbi Decoder can be fit into the StarCore processor. Considering only the transmitter, the maximum sized partition consists of all the nodes. The sum of their execution time as computed from the Table 6.1 is 5.0043. This meets the timing constraint.

7 Partitioner – Implementation Details

7.1 Introduction

The partitioning algorithm explained in the previous chapters has been implemented as command-line software on the UNIX/Linux platform. It is more likely that partitioning will be a part of a larger tool. Thus, to be useful, any implementation should be flexible enough to accommodate different structures, algorithms and constraints of the larger design flow tool. To achieve this flexibility, Object Oriented Design (OOD) principles [19] were used wherever possible while implementing the *partitioner*. The code itself was written in a platform independent structure using the C++ programming language. Thus the tool can be extended to meet additional requirements like, for example, applying more constraints on the partitions. The following sections explain the program design and the C++ implementation. At least a basic knowledge of C++ will be required to understand them.

First a few terms have to be defined. These are used in later sections while describing the design. Knowing them will help in understanding the design better.

7.2 Definition of Technical Terms

Node

A Node represents a process on the Process Model Graph.

Edge

An Edge represents the signals that are connected to input and output ports of a process.

Entity

The architecture on which a process is intended to be implemented is called the entity. For example, if a process is to be implemented as embedded software on the *StarCore* processor, then *StarCore* is the entity. Other entities include ASICs (hardware) and RISC processors (software).

Linked List

A linked list is a software representation for storing one or more elements such that certain operations can be performed efficiently on them. Each element points to the next and/or the previous element. To reach a particular element, the list is traversed starting at the head. Insertion and deletion of elements can be done efficiently in a list. Figure 7.1 shows a graphical representation of a linked list.

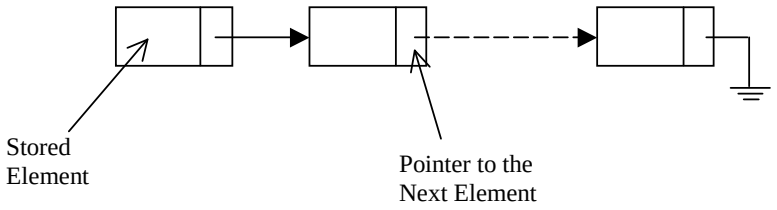


Figure 7.1 Illustration of a Singly Linked List Data Structure

Binary Tree

A binary tree is another storage representation where each element points to two other elements, also know as child elements. This forms a tree structure as illustrated in Figure 7.2. A special case of a binary tree where one child has a value lesser than and the other greater than the value of the parent is very useful in algorithm implementations. Such a binary tree allows some very efficient operations on its elements. For example, to reach any element in the tree, only one of the many paths has to be traversed, starting from the root. This can result in a very optimal implementation when the number of elements stored tends to be large.

Graph

A graph is a data structure that represents a set of nodes that are connected by a set of arcs or edges. Any node connected to a given node is called its adjacent node. Adjacent nodes can also be seen as children of the given node. The binary tree data structure is a specific form of a graph. There are several schemes to store the connection information between nodes. One of them, known as *adjacency list*, is used in this research to represent Process Model Graphs. This is explained further later in this chapter.

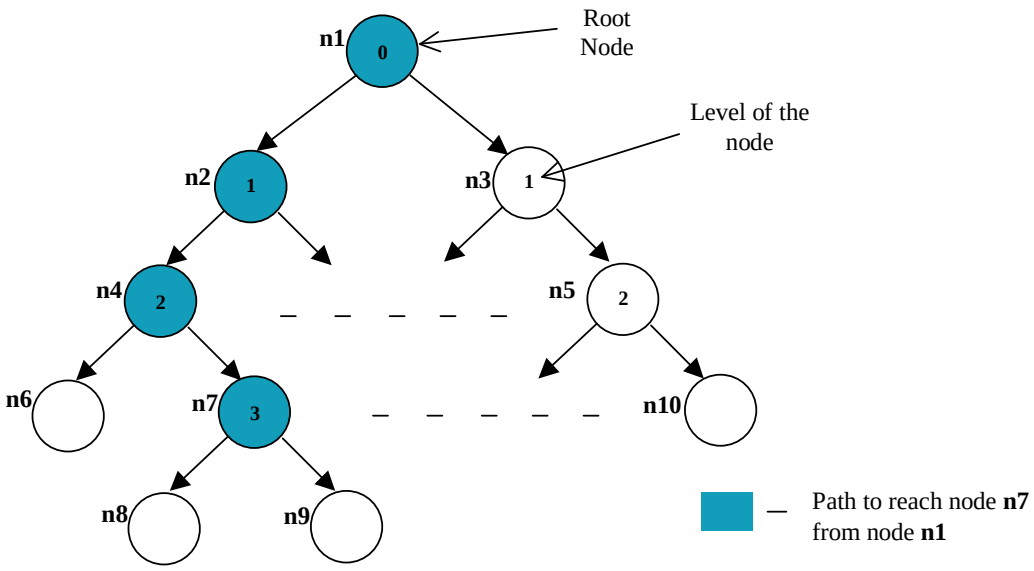


Figure 7.2 Illustration of a Binary Tree Data Structure

Depth First Search (DFS)

DFS is a method of traversing a graph where a node's children are visited before the node itself. For the graph (binary tree) shown in Figure 7.2, a DFS traversal would visit the nodes in the following order: **n6, n8, n9, n7, n4, …… n1**. For more detailed analysis see [20].

Breadth First Search (BFS)

BFS is another traversal method for a graph where a node is visited before its children. This results in an ordering where all the nodes of one level are visited before the next deeper level. BFS traversal order for the graph in Figure 7.3 is **n1, n2, n3, n4, n5, n6, n7, n10, n8 and n9**. The nodes that have not been shown in the figure have been ignored in this ordering. See [20] for more information.

Critical Path

In the Process Model Graph, there can be more than one possible path between the input port of a process and the output port of another process. Each path may contain

one or more edges (signals) and nodes (processes). If the path length is defined as the sum of the delays associated with all the edges and nodes in a path, then the critical path is the path with the maximum length.

Standard Template Library (STL)

STL is a generic C++ library that provides functionality for creating data structures like lists and binary trees. The library also provides interfaces for operating on these data structures in an efficient manner. For example, a linked list can be created and elements added to it sequentially. Later, the list can be sorted and traversed from the head or the tail of the list. In this library, the binary tree is called a *map*. More details on STL can be found in [21].

Object

An Object in the Object Oriented Design terminology is an abstraction that encapsulates an independent part of a design and all the behavior associated with it. The object also stores the state information within it. In C++, this abstraction is implemented as a Class and used as object variables.

Method

Methods are interfaces of an object that are used to interact with it. Data that is protected within an object is not visible from the outside. Methods are the only way to access or modify them. Methods are also used to implement functionality associated with an object.

7.3 C++ Class Design

This section explains the C++ implementation of the major objects used in the partitioner. This consists of a brief introduction to the functionality of the class, explanation of the methods implemented for the class and a list of some of the public and private data members. This is by no means an exhaustive design document. The interested reader is advised to look at the well-commented source code for more information.

7.3.1 UniqueId Object

To make computations efficient, all the nodes, edges and entities are represented with integer IDs. The UniqueId object implements methods that store a text string and return an integer ID corresponding to the string. If the string was already stored, the same ID is returned. Similarly the reverse functionality is also implemented. That is, the original text string can be obtained using the integer ID.

7.3.2 Node Object

As defined in the previous section, Node represents a process in the PmG. It is implemented as a data object that will hold all the attributes of a process. Some of the important elements that are encapsulated into the *Node* object are listed below

- List of all child nodes
- Pointers to the edges that are connected to this node
- List of attributes specific to each entity that this node can be implemented on. For example, the IO delays depend on the architecture on which this node is implemented and also the complexity of the node.
- Pointer to the hierarchical *PmgModule* (explained later), if this node is a hierarchical node.

7.3.3 Edge Object

Like a Node object, *Edge* is also a data object. Thus, behavioral methods are not defined for it. It stores data related to each signal in the PmG. This includes

- The activation rate of the signal
- List of nodes that are connected to this edge

7.3.4 PmgModule Object

The PmgModule object represents a major portion of the whole PmG. As explained in earlier chapters, a PmG may be either flat or hierarchical in nature. In case of a flat PmG, only the top-level module exists. On the other hand, a hierarchical PmG

consists of a top-level module and one or more hierarchical modules. Each module (top-level or hierarchical) is a complete PmG by itself. That is, it consists of a connected graph made of nodes and edges.

As the name implies, the *PmgModule* object abstracts any module of the PmG. Thus it encapsulates the set of nodes and edges of the module and their connection information. This class also implements functionality specific to the PmG module, like loading the module from a text file and computing the complexities of the nodes.

Data members of this object include

- Name of the module
- Input and Output port information for the module (if it is a hierarchical one)
- List of Nodes and Edges in the module
- List of hierarchical nodes in the module
- Adjacency list for the graph represented by this module
- Sum Complexity of the module (if this is a hierarchical module)

7.3.5 PmG Object

The PmG object completes the representation of the entire PmG. It stores a pointer to the top level *PmgModule*. If the PmG is a hierarchical one then the top-level *PmgModule* has pointers to the hierarchical *PmgModule* in its level. Each of these modules has pointers to hierarchical modules in their respective levels and so on. Thus, the entire PmG is represented as a tree structure.

The *PmG* object provides methods that operate on the entire PmG, like functions to load all the PmG modules from a text file and add or remove a partition in the PmG. Data members include

- Pointer to the top-level *PmgModule*
- List of partitions
- List of PmG modules that had errors while loading

7.3.6 Adjlist Object

The Adjlist object is a very important class that represents the adjacency list of a PmG module. Several useful functions are abstracted in this object. The implementation is a typical one for this data structure. It consists of an array whose size equals the number of nodes, with each element representing a node. The nodes that are adjacent to a given node are stored in a separate list and a pointer to the head of the list is stored in the main array. Figure 7.3 illustrates this implementation.

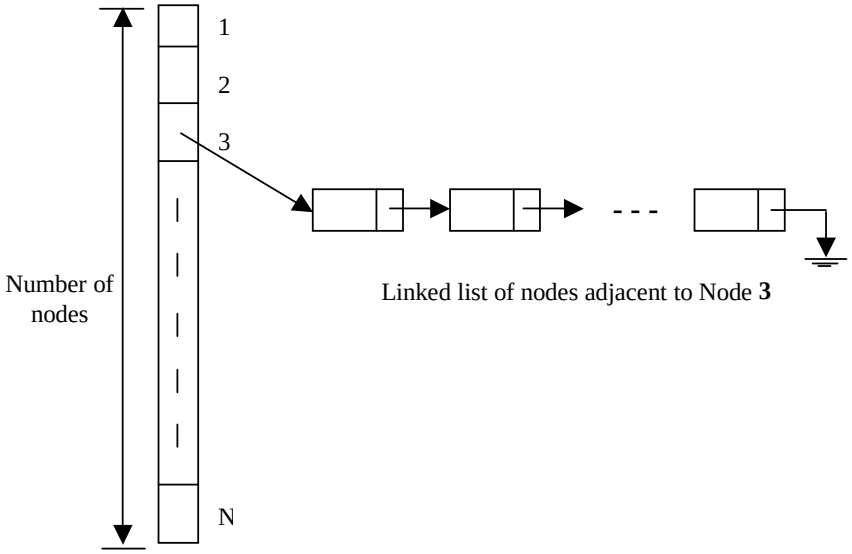


Figure 7.3 Illustration of Adjacent List Data Structure

Since the partitioning algorithm consists of choosing an adjacent node that has a preferred 'selection factor', it would only be prudent to keep the adjacency list sorted based on the selection factor. Once a node is selected for an entity, that node should no longer be considered for the rest of the selection process. A flag in each adjacency node is used to ensure this.

The adjacency list is also used for critical path computation. The BFS or DFS used for this purpose require that the graph be a directed a-cyclic graph. Thus the adjacency list object should be made more generic to support directed as well as non-directed graphs.

The *Adjlist* object has been implemented taking into account the requirements mentioned above. The object provides the following methods

- *Add* – add an adjacent node
- *Remove* – remove an adjacent node
- *Update* – sort the list based on the computed selection factor for an entity.
- *Next* – get the next adjacent node for a given node. This method remembers the state. The interface also has a parameter to select whether the graph should be considered as a digraph or an undirected one.
- *Reset* – reset all internal states

7.3.7 Partitioner Object

As the name implies, the Partitioner object implements the partitioning algorithm. It uses the other objects described in the previous sections. The following methods are implemented.

- *Startnodes* – Set a list of nodes from which the partitioning process starts. If no start nodes are specified, the algorithm executes beginning with every node in the graph.
- *Run* – This method takes three parameters, namely a reference to the PmG object, the threshold value for partitioning and the target entity for partitioning, and runs the greedy algorithm for partitioning the PmG.
- *Constrain* – Constrain the partitions based on an attribute, for example, buffer size or Power dissipated. The attributes for each node for a given entity should be defined along with the PmG.
- *Max* – List the maximum sized partitions
- *All* – List all partitions found

7.3.8 The Language Parser

The BNF form of the language used to describe the PmG text file is given in Appendix A. The parser used for syntax verification was written using the popular Unix tool called Yet Another Compiler Compiler (YACC). This tool generates a C

language implementation of a state based parser, which is invoked from the *PmgModule* object. Each PmG module passes the parsing information to the parser through the *ParseData* object. The information includes the name of the text file corresponding to the module and pointers to the list of nodes and edges. The parser reads the text file and creates the list of nodes and edges for the file and updates the pointers in the *ParseData* object. If the file had syntax or semantic errors, they are also updated in that object.

7.4 Code Integration

The objects described so far form pieces for the overall partitioning tool. These pieces have to be integrated to obtain a usable functionality. For instance, the PmG should be loaded from its text files and checked for errors. Any deadlines present in the graph should be specified. Then the partitioning tool must be run, possibly multiple times, for different entities with corresponding threshold values. All constraints have to be applied on the obtained partitions to get the final list of acceptable partitions.

The CLI object does this integration by providing a Command Line human Interface. Several commands have been implemented to partition the graph as well as to obtain useful statistics. This object is briefly explained below.

7.4.1 CLI Object

The CLI module manages the user interactions by implementing several useful commands. At the core of the CLI is a loop that displays a command prompt and waits for the user to type a command. The command is then interpreted, executed and the results displayed. After this cycle, the loop goes back to the command prompt and waits for another user command.

The CLI Object stores other objects within it. The PmG Object for example, is a private data member of this class. A pointer to it is stored and the object is dynamically created when the PmG is loaded from the text file. Once created, this PmG object is used for other operations like setting deadlines, computing the critical path and is also passed as a parameter to the partitioning routine.

The *Partitioner* object is also a member in the CLI class. Like the PmG Object, this object is not created until it is actually used when the partition command is issued.

The following are some of the important commands implemented.

Load Command

The Load command is issued to read a PmG from its text file and load it into the PmG Object. The full path to the top level PmG module file is passed as the command argument. As mentioned earlier, this causes a *Pmg* object to be created. Any previously loaded *Pmg* object is first destroyed. Syntax or semantic errors found in the PmG file are reported to the user. Additionally, as a part of loading, complexity computations are done for each node. For hierarchical nodes, the sum complexity of all the nodes of the child module is also computed.

Partition Command

The Partition command operates on the *Pmg* object and partitions it for the specified entity. As mentioned earlier, the partitioning algorithm is implemented in the *run* method of the *Partitioner* Object.

This command requires the name of the target entity as the argument. This is used to select complexity values corresponding to the entity. Optionally, the threshold for the partitioning algorithm can also be specified. By default, the threshold is one. But sometimes a lesser value may be specified, if the target entity cannot be used to its full capacity. This may be because of switching delays in software entities or other environment considerations.

If there is a specific set of start nodes for the algorithm, they have to be set using a separate command before invoking the partition command. Otherwise, the algorithm executes once starting at every node.

The result of this command is a list of all feasible partitions that were found. Other commands can be used to limit this list to only the largest sized partitions or only partitions that are constrained by certain attributes.

Critical Paths Command

This command finds all the critical paths between two specified ports. The PmG object is needed to compute the critical paths. Since the critical paths have to be computed on the fly for any set of ports, the CLI object does not have an instance of the critical path object. Instead, every time the critical path command is invoked, an instance is dynamically generated and the paths are computed.

Other commands

Other commands that have been implemented in the *CLI* object include

- List all the nodes
- List all partitions
- List complexities for each node
- Set deadlines
- Check if all deadlines have been met
- Set threshold value

For more information about these commands, see the comments in the source code. Figure 7.4 shows an execution flow chart for the CLI object.

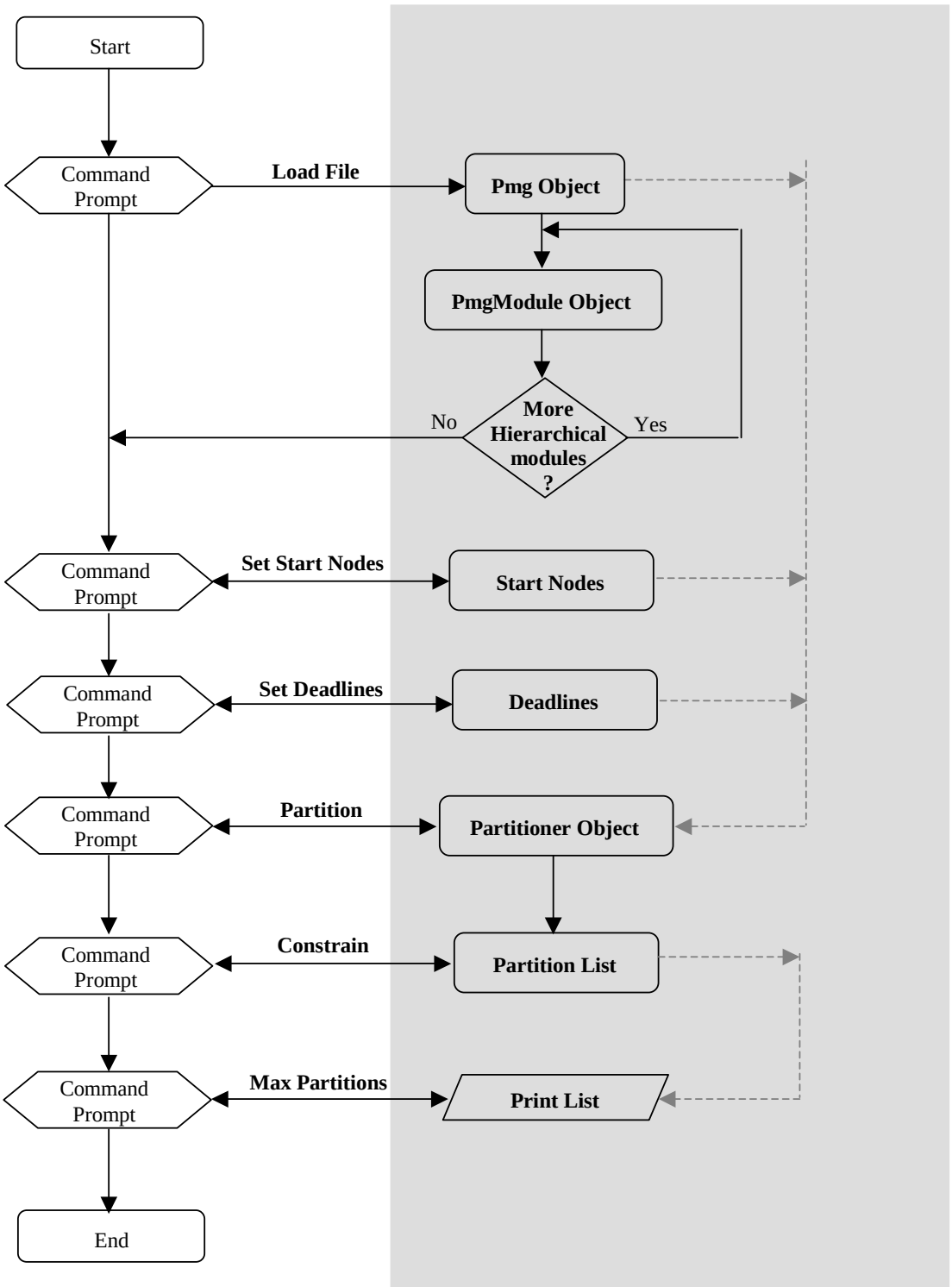


Figure 7.4 Interaction between CLI Objects Methods and Data members

7.5 Summary

The design of the tool has been kept very simple. The goal was to study the algorithm more than to produce a high performance application. While performance has been considered to some extent, the tool may not be suitable for graphs with a very large number of nodes and edges.

The *partitioner* program has been written with an aim of keeping it more readable and easily expandable. The idea was to be able to integrate it into a larger design flow tool with very few modifications. Object Oriented Design was the obvious choice. At the heart of the program is the CLI Object that provides a command line interface to the user. Commands issued by the user results in the dynamic creation of other objects that perform the partitioning action.

To achieve integration with other tools, the node and edge data structures of any tool have to be derived from the node and edge objects that are implemented in the partitioner. With a few other issues handled, the partitioner object can be made to work with PmGs loaded from other tools.

8 Conclusion and Future Work

As System On a Chip (SoC) gains wider acceptance, designers are moving towards a model based design flow. In a model-based scheme, the entire system is modeled and verified before implementation decisions are made. This results in reduced cost and faster design to market time. However, such a co-design approach has its own challenges in the area of verification and partitioning. This research made an attempt to address the latter issue, namely partitioning. A process model graph that is extracted from a system level model was used as the basis for the partitioning scheme proposed.

Graph partitioning is an NP-Hard problem. To find near-optimal solutions calls for heuristic based approaches. This thesis proposed a greedy algorithm for graph partitioning using a complexity heuristic. The algorithm was implemented as a software tool and tested on a model of the GSM communication system. The results were presented in Chapter 6.

The partitioning utility supports a few useful commands that operate on the PmG. However, there are some functional limitations in the tool. For instance, cyclic graphs that result from control flow operations like loops are not supported. In future, functionality can be expanded to support control flow graphs as well.

Even though the present implementation gave reasonable results, it has an inherent weakness. The greedy algorithm, though quite fast, does not produce the most optimal results. Where a very tight partitioning is required, more advanced partitioning algorithms have to be used. The complexity measure derived in chapter 4 can be incorporated into the cost function of a more sophisticated partitioning scheme.

9 References

1. Klaus Buchenrieder and Jerzy W. Rozenblit. “Codesign: Computer-Aided Software/ Hardware Engineering”, *IEEE Press*, 1995
2. Dr. George Wang. “The Path to System-On-Chip”, Slide Show, June 29, 1999
3. Gerard Berry. “The Esterel v5 Language Primer”, *Centre de Mathématiques Appliquées*, July 2000
4. M. Chiodo, et. al, “Hardware- Software Codesign of Embedded Systems”, *IEEE Micro*, Vol. 14, No. 4, pp.26-36, Aug. 1994
5. Wooseung Yang, et. al., “Current Status and Challenges of SoC Verification for Embedded Systems Market”, *Proceedings. IEEE Internations, SOC Conference*, Sept 2003
6. James R. Armstrong and Gail F. Gray. “VHDL Design – Representation and Synthesis”, *Prentice Hall*, Second Edition, 2000
7. Stuart Swan. “An Introduction to System Level Modeling in SystemC 2.0”, *Open SystemC Initiative*, White paper, May 2001
8. Thorsten Grötker, et. al, “System Design With SystemC”, *Kluwer Academic Publishers*, May 2002
9. J.R. Armstrong, P. Adhipathi, J.M. Baker, Jr. “Model and Synthesis Directed Task Assignment for Systems On A Chip”, *Parallel and Distributed Computing Society*, 2002
10. Amer Baghdadi and et. al, “Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting Multiprocessor Systems”, *IEEE Transactions on Software Engineering*, Vol. 28, No. 9, September 2002
11. M. Nemani and F. N. Najm, "Delay estimation for VLSI circuits from a high-level view," *35th Design Automation Conference*, pp. 591-594, San Francisco, CA, June 15-19, 1998
12. C. Carreras et. al., “A Co--Design Methodology Based on formal Specification and High--Level Estimation”, *IEEE Proc. of Fourth International Workshop on Hardware/Software Codesign*, Pittsburgh, Pennsylvania, 1996
13. A. Srinivasan and et. al, “Accurate area and delay estimation from RTL descriptions”, *IEEE Transactions on VLSI Systems*, 6(1):168--172, Mar. 1998

14. A.A. Jerraya, et al., “Multilanguage Specification for System Design and Codesign” chapter on “System-Level Synthesis,” *NATO ASI 1998*, A. Jerraya and J. Mermet eds., Kluwer Academic, 1999
15. K. Suzuki and A. Sangiovanni-Vincentelli, “Efficient Software Performance Estimation Methods for Hardware/Software Codesign”, *Proceeding of Design Automation Conference*, June 1996
16. J. Bammi, et. al., “Software Performance Estimation Strategies in a System-Level Design Tool”, *Proc. CODES*, May 2000
17. A. Varma. “Modeling and Synthesis with SystemC”, *Bradley Department of Electrical Engineering*, Master of Science thesis, Virginia Tech, 2001
18. B. Sirpatil. “Software Synthesis of SystemC Models”, *Bradley Department of Electrical Engineering*, Master of Science thesis, Virginia Tech, 2001
19. James E. Rumbaugh, et. al., “Object-Oriented Modeling and Design”, *Prentice-Hall*, 1991
20. Alfred V. Aho, et. al., “Data Structures and Algorithms”, *Addison-Wesley Pub Co*, January 1983
21. Alexander Stepanov and Meng Lee, “The Standard Template Library”, *ANSI/ISO document*.
22. Ali Sayinta, et. al., “A Mixed Abstraction Level Co-Simulation Case Study Using SystemC for System on Chip Verification”, *Proceedings of the Design,Automation and Test in Europe Conference and Exhibition (DATE’03)*, IEEE, 2003
23. Wayne Wolf, “A Decade of Hardware/Software Codesign”, *IEEE Computer*, V36, No4, 38-43, April 2003
24. Peter Arato et. al, “Hardware-Software Partitioning in Embedded System Design”, *IEEE International Symposium on Intelligent Signal Processing*, 197-202, Sept. 2003
25. Karthikeyan Bhasyam, et. al, “HW/SW Codesign Incorporating Edge Delays Using Dynamic Programming”, *Proceedings of the Euromicro Symposium on Digital System Design*, IEEE Computer Society, Sept. 2003
26. GUITTON-OUHAMOU Patricia, et. al, “Energy Optimization in a HW/SW Tool: Design of Low Power Architecture System”, *Proceedings of The 3rd IEEE International Workshop on System-on-Chip*, IEEE Computer Society, 2003

Appendix A – BNF Representation of PmG

The grammar that defines the PmG textual representation is given below. The Backus Naur Form (BNF) notation with slight modification has been used to present the rules.

```
file            : header sigdecl prodefn
                ;

header          : main_header
                | hier_header
                ;

main_header     : MAIN_PMGID modulename
                ;

hier_header     : HIER_PMGID modulename '(' inportmap ';' outportmap ')' ';'
                ;

modulename      : IDENTIFIER
                ;

inportmap       : posmap ':' INP
                ;

outportmap      : posmap ':' OUTP
                ;

sigdecl         : SIG '{' sigstmts '}' ';' { }
                ;

sigstmts        : sigstmt
                | sigstmt sigstmts
                ;

sigstmt         : IDENTIFIER ':' VALUE ';'
                ;
```

```
prodefn         : prostmt
                | prostmt prodefn
                ;

prostmt         : profmt '{' proattrs '}' termchar
                ;

profmt          : norm_profmt
                | hier_profmt
                ;

norm_profmt     : IDENTIFIER
                ;

hier_profmt     : norm_profmt '<' modname ':' STRING '>'
                ;

modname         : IDENTIFIER
                ;

proattrs        : proelem
                | proattrs proelem
                ;

proelem         :
                | inmap
                | outmap
                | delays
                | attributes
                ;

inmap           : INP '(' prtmap ')' ';'
                ;

outmap          : OUTP '(' prtmap ')' ';'
                ;

prtmap          : posmap
                | ascmap
                ;
```

```
posmap          : IDENTIFIER
                | posmap ',' IDENTIFIER
                ;

ascmap          : mapstmt
                | mapstmt ',' ascmap
                ;

mapstmt         : IDENTIFIER '=' VALUE
                ;

delays          : delmap
                | delays delmap
                ;

delmap          : DELAY '(' IDENTIFIER ')' '{' delstmts '}' termchar
                | DELAY '{' delstmts '}' ';'
                ;

delstmts        : delstmt
                | delstmt delstmts
                ;

delstmt         : VALUE ',' VALUE ':' VALUE TMUNIT ';'
                ;

attributes      : ATTRS '(' IDENTIFIER ')' '{' attrstmts '}' termchar
                ;

attrstmts       : attrstmt
                | attrstmt attrstmts
                ;

attrstmt        : IDENTIFIER '=' VALUE ';'
                ;

termchar        : ';'
                ;
```

Terminals

IDENTIFIER	: Any alpha-numeric character string that starts with an alphabet
VALUE	: Any numeric value
STRING	: Any quoted character string
MAIN_PMGID	: The string 'MAIN_PMG'
HIER_PMGID	: The string 'HIER_PMG'
INP	: The string 'IN'
OUTP	: The string 'OUT'
SIG	: The string 'SIGNAL'
DELAY	: The string 'DEL'
ATTRS	: The string 'ATTRIBUTES'
TMUNIT	: Any of the following strings 'NS', 'ns', 'US', 'us', 'MS', 'ms'